GRAPH-BASED PARALLEL QUERY PROCESSING AND DATA PARTITIONING IN OBJECT-ORIENTED DATABASES

BY

YAW-HUEI CHEN

A DISSERTATION PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

UNIVERSITY OF FLORIDA

1993

To my parents

ACKNOWLEDGEMENTS

I am deeply indebted to Dr. Stanley Y. W. Su, chairman of my supervisory committee, for his continuous guidance, help, and support throughout my doctoral study. I am grateful to Dr. Herman Lam, Dr. Randy Chow, and Dr. Theodore Johnson for their comments and suggestions on this work. I thank Dr. Sheng S. Li for being on my supervisory committee and for attending my defense on short notice. I would also like to thank Dr. Jack R. Smith for his support and for being on my supervisory committee over a long period of time.

I thank Sharon Grant, the secretary of the Database Systems Research and Development Center, for her help and support. I am thankful to my colleagues and friends for their stimulating discussion and invaluable friendship during my stay in Gainesville.

My special thanks go to my wife Lin Ching for sharing the burdens of these years, and my son Bryan for bringing so much joy into my life.

This research was partially supported by National Science Foundation and Fujitsu, Limited.

TABLE OF CONTENTS

Abstract of Dissertation Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Doctor of Philosophy

GRAPH-BASED PARALLEL QUERY PROCESSING AND DATA PARTITIONING IN OBJECT-ORIENTED DATABASES

By

Yaw-Huei Chen

December 1993

Chairman: Dr. Stanley Y. W. Su
Major Department: Computer and Information Sciences

Object-oriented database management systems (OODBMSs) provide rich facilities for the modeling and processing of structural as well as behavioral properties of complex application objects. However, due to their inherent generality and continuously evolving functionalities, efficient implementations are important for these OODBMSs to support the present and future applications, particularly when the databases are very large.

In this dissertation, we present several parallel, multi-wavefront algorithms based on two processing approaches, i.e., identification and elimination approaches, to verify association patterns specified in queries. Both approaches allow more processors to operate concurrently on a query than the traditional tree-structured query processing approach, thus introducing a higher degree of parallelism in query processing. A graph model is used to transform the query processing problem into a graph problem. Based on the graph model, proofs of correctness of both approaches for tree-structured queries are given, and a combined approach for solving

cyclic queries is also provided and proved. A heuristic method is also presented for partitioning an OODB. The main consideration for partitioning the database is load balancing. This method also tries to reduce the communication time by reducing the length of the path that wavefronts need to be propagated. Multiple wavefront algorithms based on the two approaches for tree-structured queries have been implemented on an nCUBE 2 parallel computer. The implementation of the query processor allows multiple queries to be executed simultaneously. This implementation provides an environment for evaluating the algorithms and the heuristic method for partitioning the database. The evaluation results are presented in this dissertation.

CHAPTER 1
INTRODUCTION

Traditional database systems are mainly designed for supporting business applications. They cannot provide the needed modeling capabilities and system functionalities for many new data-intensive applications, such as engineering, science, medicine, multimedia, and military. Object-oriented database management systems (OODBMSs), which provide better structural and behavioral abstraction facilities, offer a promising alternative to manage data in these new domains [Su89, Deux90, Kim90, Wilk90]. However, due to their inherent generality and continuously evolving functionalities, efficient implementations are important for these OODBMSs to support the present and future applications, particularly when the databases are very large.

The performance of large OODBs is often limited by the sequential nature of the conventional computer systems in which the OODBs and OODBMSs are implemented. Several opportunities for parallelism inherent in many OODBMSs, such as inter-query parallelism, inter-operator parallelism, and intra-operator parallelism, cannot be easily exploited by the uniprocessor von Neumann computer architecture. In addition, the I/O bottleneck found in the conventional computer system degrades the performance of OODBs very much. One obvious solution to this I/O bottleneck problem is to increase the I/O bandwidth through parallelism. Thus, in order to improve performance, parallel algorithms and architectures are needed to support OODB processing.

Many parallel execution strategies for relational database operations, especially for the expensive join operation, have been proposed in the literature [Vald84, Schn89, Kits90], but most of the work has addressed the problem of executing joins involving only two relations. Recently, researchers have studied parallel execution strategies for multi-way join queries using different query representations such as left-deep, right-deep, and bushy trees [Schn90, Lu91]. When many relations and operations are involved in a query, the traditional tree-based query processing has two potential problems: (1) it may generate large intermediate results; (2) the tree representation implies an inherent bottom-up evaluation order.

In distributed query processing, the semijoin operator is used to reduce the sizes of relations and/or intermediate results [Bern81]. Most research efforts have focused on the problem of finding the optimal execution order of semijoins to reduce the data transmission cost [Aper83, Chen89, Chen91a]. However, this tree-based execution of semijoins may lose the opportunity for exploiting some types of parallelism, such as inter-operator parallelism and inter-query parallelism. New semijoin execution strategies have been proposed to improve the performance by executing all applicable semijoins of relations at the same time [Wang91] and by adding a backward reduction phase to the semijoin and pipelining the execution [Rous91]. Since all reduced relations and/or intermediate results are sent to a final site to perform a multi-way join, a large amount of data is transmitted in the network and the final site may eventually be overwhelmed by the incoming data and become a bottleneck in a distributed query processing. As pointed out in Jenq et al. [Jenq90], most existing techniques developed for relational query processing can be directly applied to object-oriented queries against a single target class. For example, the tree-based query processing techniques for relational databases are also used in OODBs [Khos88, Jenq90].

In addition to the problems associated with the traditional tree-based query processing, queries in OODBs are usually rather complex due to the complex data structures used to represent object classes and objects. Also, OODB queries often involve searching and retaining multiple target classes of objects on which different user-defined operations (or methods) are performed. Therefore, we need to develop novel parallel algorithms based on general graph representations and tailor them to take advantage of the OO data and query characteristics for processing large OODBs.

In this work, all things of interest to an application (e.g., physical objects, abstract things, events, etc.) are uniformly defined as objects. An object is uniquely identified by an object identifier (OID). Objects which have the same structural and behavioral properties are grouped together to form an object class. The class defines the abstract data type for these objects and serves as a container of object instances, which are the data representations of the objects in the class. An object instance is composed of an instance identifier (IID), the descriptive data of the object instance, and references to other object instances. An IID is represented by the concatenation of OID and class ID so that object instances of the same object in multiple classes can be unambiguously distinguished. Furthermore, classes can be associated with one another through different types of associations (e.g., aggregation, generalization, etc.) and the objects of these classes are thus inter-related with one another through these associations. Therefore, the intension of an OODB can be viewed as a network of inter-connected object classes with different types of associations and the extension of an OODB can be viewed as a network of object instances inter-connected through these associations. Object instances can be accessed and manipulated by pattern specifications and verifications [Alas89, Guo91]. We call the operation of

finding the connection between two patterns of object associations the Associate operator, which constructs a new pattern of object association by concatenating two operand patterns. In the literature, several similar operators have been proposed to find related objects or tuples, such as class traversal [Jenq90, Kim90], functional join [Zani83, Care88], pointer-based join [Shek90], and assembly [Kell91]. Since a chain query (or a long path of associated object classes) and tree- or network-structured queries are commonly specified in OODB applications, it is important to have a query processor which can exploit inter-operator parallelism for queries composed of many Associate operators in a complex structure.

A two-phase query processing strategy, which consists of an identification phase and a result-processing phase, has been proposed to avoid accessing and processing a large amount of (sometimes unnecessary) data [Lam89, Thak90b]. In the first phase, objects of interest in the database are identified in the form of IIDs. Then, system- and/or user-defined functions are executed on the objects selected during the first phase in order to produce the final result. Since only IIDs are processed and propagated in the first phase and the retrieval of sizable descriptive data is postponed until the second phase when the relevant objects have been identified, this technique can reduce or eliminate the I/O bottleneck found in large OODB systems.

This dissertation presents parallel query processing techniques for OODBs. The main contributions of this work are as follows:

1. We present two pattern-based and two-phase query processing approaches, identification and elimination, for verifying object association patterns. The identification approach, which identifies objects that satisfy the given pattern, has been researched and reported in Thakore [Thak90a] and in Thakore et al. [Thak90b]. We shall describe a modified identification algorithm to serve as a

contrast to the new elimination approach. Instead of identifying objects that satisfy the specification, the elimination approach eliminates objects that do not satisfy an association pattern specification given in a query [Su91].

2. We present a formal graph model to transform the query processing problem into a graph problem. Based on the graph model, proofs of correctness of both approaches for tree-structured queries are given. Also, a combined approach for solving cyclic queries is provided and proved.

3. Parallel and multi-wavefront algorithms based on the two approaches are introduced for both acyclic and cyclic queries. These algorithms can have more processors to operate concurrently on a query than the traditional tree-based query processing approach. Thus, a higher degree of parallelism in query processing can be introduced.

4. A new data structure is introduced to store the OODB. Both identification- and elimination-based multiple wavefront algorithms can work on the same data structure. Therefore, it makes it easier to choose a suitable algorithm to process the query.

5. We propose a heuristic method for partitioning the database. Load balancing is the main consideration for partitioning the database.

6. Multiple wavefront algorithms based on the two approaches for tree-structured queries have been implemented on an nCUBE 2 parallel computer. The implementation of the query processor allows multiple queries to be executed simultaneously. This implementation provides an environment for evaluating the algorithms and the heuristic method for partitioning the database.

This dissertation is organized as follows. A survey of related work on parallel query processing and database partitioning is given in Chapter 2. In Chapter 3, we illustrate the concepts of data modeling and pattern-based query processing in OODBs using the notions of schema graph, object graph, and query graph. Then, a formal graph model is presented and the query processing problem is defined as a special subgraph mapping problem. In Chapter 4, the identification approach and the elimination approach for tree-structured queries are presented together with a combined approach for cyclic queries. Based on the formal graph model, we prove the correctness of each algorithm in this chapter. In Chapter 5, an asynchronous parallel system and the data structures used for storing objects and their associations are described. Then, parallel wavefront algorithms based on both approaches for tree-structured queries and a combined parallel algorithm for cyclic queries are presented. We also discuss in this chapter the issues of starting query processing from a selected number of classes and the implementation of inter-class comparison conditions. In Chapter 6, we propose some heuristic rules for partitioning the database. In Chapter 7, we present and evaluate the implementation of the query processor on an nCUBE 2 parallel computer. Finally, conclusions and future work are given in Chapter 8.

CHAPTER 2
SURVEY OF RELATED WORK

In this chapter, we present a survey of some existing work related to the query processing techniques introduced in this dissertation. In Section 2.1, a survey of different operators that relate objects or tuples in a database is given. Then, we present a survey of query processing and optimization techniques in Section 2.2. Finally, a survey of database partitioning is presented in Section 2.3.

2.1 Finding Related Objects

Finding related objects or tuples in a database is a fundamental operation and has been extensively studied by many researchers. For example, pointer-based joins in a relational database use either explicitly stored pointers or system maintained pointers to link tuples in different relations [Care90, Haas90, Shek90]. A pointer-based join optimization structure, join indices, maintains a pre-computed join by storing TIDs (tuple identifiers) of two relations that match a join predicate [Vald87]. Functional joins are used to find related objects in semantic and OO data models which link object instances through reference attributes or object-valued attributes [Zani83, Care88]. As pointed out in Jenq et al. [Jenq90], most existing techniques developed for relational query processing can be directly applied to object-oriented queries against a single target class.

Since an OODB query usually contains many classes instead of just two relations in most relational database queries, class traversals have been proposed to find the "joining" order of the classes in a query graph [Jenq90, Kim90]. Also, an assembly operator has been proposed to translate a set of complex objects from

their disk representations to a quickly traversable memory representation [Kell91]. However, similar to semijoin operations [Bern81], these traversal and assembly operators traverse the query graph up and down like traversing a tree. This tree-based approach has an inherent drawback which is the implied sequential bottom-up evaluation, and thus its ability of exploiting the inter-operator parallelism is limited. Recently, researchers have begun to understand the alternative query tree organizations in various database environments [Hara92, Ioan91, Lu91, Schn90]. For example, experiment results demonstrate that right-deep tree scheduling strategy provides more opportunities for exploiting parallelism than the left-deep tree strategy in a multiprocessor database system. In addition to the problems associated with the tree-based query processing, queries in OODBs are usually complex and involve multiple target classes, and therefore novel parallel algorithms based on general graph representations are needed for large OODBs.

2.2 Query Processing and Optimization

Most existing works on parallel database processing have addressed the problem of executing joins that involve two relations. For example, four multiprocessor join algorithms have been implemented on Gamma: sort-merge, Grace [Kits84], Simple [DeWi84, Shap86], and Hybrid [DeWi84, Shap86], where the Hybrid join algorithm is the default join strategy [Schn89]. All four algorithms are based on the concept of hash-based partitioning in which the two relations to be joined are partitioned into a set of disjoint buckets by applying the same hash function. This form of parallelism is called intra-operator parallelism and requires data partitioning.

The principle of dataflow systems [Denn80, Gaud91, Trel82] has been applied to many multiprocessor database machines. Dataflow scheduling techniques are used in Gamma to coordinate multioperator queries [DeWi86, DeWi90]. The asynchronous

dataflow model is the basis of the distributed execution model of Bubba [Alex88]. During the data-driven query processing in AGM [Bic89], tokens propagate asynchronously through the network in search of results satisfying the given query. Other researchers also utilize the dataflow techniques to exploit the pipelining parallelism in query processing [Grae90, Wils91].

The query optimization research studies techniques and strategies that can be used to improve the efficiency of query evaluation procedures. In general, exact optimization of query evaluation procedures is computationally intractable [Chan77, Ibar84]. Algebraic manipulation and cost estimation are two basic kinds of query optimization strategies. The algebraic manipulation strategies logically transform the query representation to reduce the cost of answering the query. The second class of strategies consider issues, such as the existing storage structures and access paths, and a cost model, to select from among alternatives the query evaluation plan (QEP) that is best for the existing data and structures. There are many surveys of the query optimization research in general in the literature [Jark84, Kim85, Maie83, Ullm89, Yu84].

One of the common heuristics used for query optimization in algebraic manipulation is moving selection and projection operators as far down the parse tree as possible [Hall76, Smit75]. Since it may be beneficial to share common subexpressions during execution, some studies have focused on how to identify them within a single query or among a set of queries [Chak86, Fink82, Hall76, Jark85], and other studies have emphasized the sharing of common intermediate results or common tasks in the process of query evaluation [Chak91, Park88, Rose88, Sell88, Su86].

Graph representations have been used in query processing and optimization. For example, operations of the decomposition algorithm [Wong76] can be viewed as removing hyperedges and nodes from the connection hypergraph of the query

[Maie83, Ullm89]. As it decomposes the hypergraph, the algorithm generates a program that will compute the answer for the query. The connection hypergraph decomposition algorithm has been extended to generate a single program for evaluating a set of queries [Chak86].

Query optimization is an expensive process, primarily because enormous alternative QEPs exist for a query. For example, the algorithm described in System/R [Seli79] compares all possible join orders to find the optimal execution. Since the space requirement is exponential (i.e. $O(2^N)$ where N is the number of relations in the query [Seli79]), this algorithm is not feasible when the number of relations in a query is more than ten. Ibaraki and Kameda [Ibar84] prove that the problem of minimizing the number of page fetches necessary to evaluate a given query using the nested-loop join method is NP-complete. However, if the query is a tree query and the cost function satisfies a certain form, they provide an algorithm that can find the optimal sequence for evaluating the query in $O(N^2 \log N)$ time. An improved $O(N^2)$ algorithm for optimizing non-recursive queries has been reported in Krishnamurthy et al. [Kris86].

For solving the multi-way join (or large join) query optimization problem, several heuristics, such as iterative improvement, simulated annealing, and two-phase optimization, have been proposed in the literature [Ioan87, Ioan90, Swam88, Swam89]. Through a combination of analytical and experimental analysis, it is found that the shape of the cost function of all QEPs of general join processing trees (bushy trees) resembles a "well" [Ioan91]. Therefore the strategy space of bushy trees is easy to optimize and heuristics like the two-phase optimization can produce good and stable output. Different types of query plan structures, left-deep, right-deep, and bushy, in processing multi-way join queries in shared-nothing architecture have been studied by Schneider and DeWitt [Schn90]. Lu et al. [Lu91]

proposed greedy optimization algorithms for generating multi-way join query plans in a shared-memory multiprocessor system. The multi-way join query plans are essentially operator trees and are evaluated in a leaf-to-root order. In this research, the query processor can start evaluating the query graph from every node, thus allowing a higher degree of inter-operator parallelism in query processing.

Although many techniques have been introduced for parallel database processing and query optimization, they have been used in isolation in separate systems. We believe that it is necessary to integrate these techniques together with some new ones to be introduced in this research in order to achieve the efficiency needed for supporting very large OODBs.

2.3 Partitioning a Database

In order to achieve parallelism, the database needs to be partitioned over multiple components in a parallel system. For example, relations in Gamma [DeWi86, DeWi90] are horizontally partitioned across all nodes with disk drives using one of four declustering strategies provided in the system: round-robin, hashed, range, and hybrid-range partitioned. However, none of the strategies is a clear winner in the performance analysis [Ghan90b, Ghan90a]. To decluster all relations across all nodes with disks is recognized as a serious mistake [DeWi90]. A better solution used in Bubba [Cope88] is to decluster a relation based on the "heat" (i.e., the cumulative access frequency) and the size of the relation. Since the ideal data placement changes continuously as the workload changes in time, Bubba repeatedly refines the data placement if the performance improvement is worth the work required to reorganize.

In a relational database environment, a relation may be accessed by several types of queries which require different sets of attributes. In order to improve the

performance, attributes of the relation are divided into groups and the relation is projected into fragment relations according to these attribute groups. This process is called vertical partitioning. The fragments are assigned to different sites in distributed database systems to minimize the cost of accessing data by all queries.

There are trade-offs between horizontal and vertical partitioning methods. A general discussion of pros and cons on a decomposed storage system (DSM), which pairs each attribute value with the surrogate of its record, is reported by Copeland and Khoshafian [Cope85]. Several parallel database projects have employed some form of the same vertical data partitioning concept [Khos87, Khos88, Lam87, Vald87].

As for the OODBs, it is recognized that object clustering is important to the performance [Chen91b, Shan91, Tsan91]. However, the clustering in OODBs is still an open research issue, and therefore the problem of declustering an OODB for a parallel system is a new challenge in research.

CHAPTER 3
OBJECT-ORIENTED DATABASES AND QUERY PROCESSING

An object-oriented data model has the following features: object identity, data abstraction, encapsulation, and inheritance of structure and behavioral properties. A database modeled in an object-oriented data model can be represented by graphs. This graphical view of OODBs leads to a pattern-based query model in which a query can be formed and processed by specifying and manipulating association patterns among objects in the database. An overview of OODBs and their pattern-based query model are described in this chapter. A formal graph model is also presented so that the query processing problem can be transformed into a graph problem. The conceptual understanding of the formal graph model is necessary for the comprehension of query processing algorithms which will be presented in the subsequent chapters.

3.1 The Graphical View of Object-oriented Databases

In an object-oriented world, all things of interest to an application, such as physical entities, abstract concepts, events, processes, or functions, are uniformly defined as objects. Objects having the same structural and behavioral properties are grouped together to form an object class which defines the abstract data type for these objects. The behavioral properties of an object class are defined in terms of system-defined (e.g., retrieve, update, etc.) and/or user-defined (e.g., purchase a part, hire an employee, etc.) operations. The structural properties of an object class are the descriptive data (or attributes) of objects in the class and the association data of this class with other classes. Different types of associations can be used

to specify the relationships between classes. The two most commonly recognized associations are aggregation and generalization. Aggregation models the a–part–of, a–function–of, or a–composition–of relationship, and generalization models the is–a or superclass–subclass relationship. In general, classes can be associated through different types of associations and object instances of these classes are inter-related with one another through these associations.

Any OODB which encompasses the common structural properties described above can be depicted as two graphs: a schema graph and an object graph. In the schema graph (i.e., the intensional database), the database is viewed as a collection of object classes inter-related through various types of associations. The schema graph of an example university database is shown in Figure 3.1. Rectangle vertices represent entity classes and circle vertices represent domain classes. Objects in entity classes are entities of interest in an application domain. Each object is assigned with a system-wide unique OID. Objects in a domain class are self-naming (e.g., integer 2, age 5, etc.) and serve as values for defining or describing other entity or domain class objects. Edges in the schema graph are bi-directional and represent associations among object classes. The edges labeled by G (for generalization association) denote superclass-subclass relationships between classes. The edges labeled by A (for aggregation association) denote descriptive attributes or object references.

In the object graph (i.e., the extensional database), the database is viewed as a collection of objects, grouped together in object classes, and inter-related through type-less associations. The associations are type-less at the extensional level because once a schema has been defined, the semantics of the association types become constraints of object manipulations enforced by the OODBMS and the user does not have to be concerned about them when formulating queries. Vertices in the

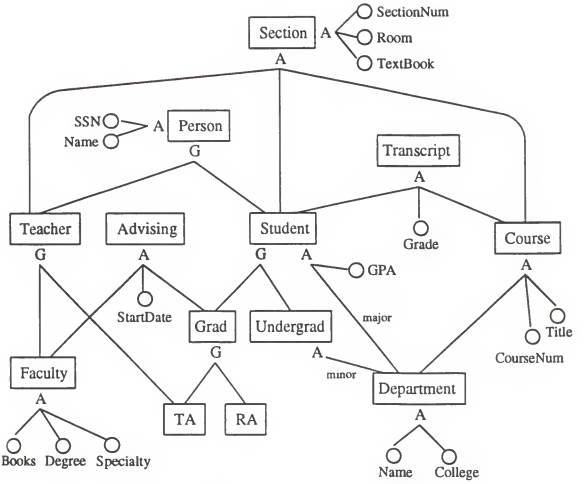


Figure 3.1 Schema graph of a university database.

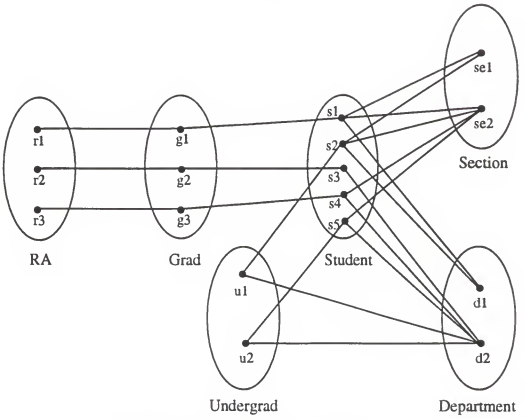


Figure 3.2 A subdatabase's object graph.

object graph represent object instances in the database, and each edge represents the association between two inter-related object instances.

A part of the university database which contains classes RA, Grad, Undergrad, Student, Section, and Department is used as an example subdatabase throughout this work. One example object graph of this subdatabase is shown in Figure 3.2. In this example, the Student class contains five object instances: s1, s2, s3, s4, and s5. Student object instance s1 is associated with Grad object instance g1 which, in turn, is associated with RA object instance r1; representing that student s1 is a graduate student and holds an RA position. Student object instance s1 is also associated with Section object instances se1, se2 and Department object instance d1. This represents that student s1 majors in department d1 and currently takes courses offered as sections se1 and se2. Note that Student object instance s3 is not

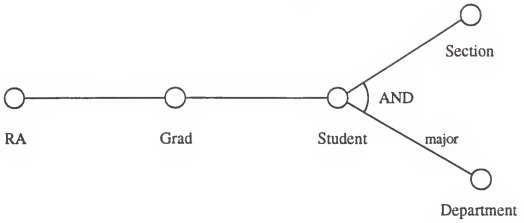


Figure 3.3 Query graph of Query 1.

associated with any Section instance, which means that student s3 does not take a course currently.

3.2 The Pattern-based Query Processing

Based on the above graphical view of OODBs, an object-oriented query language [Alas89, Alas90] and its underlying algebra have been introduced in which a query can be formed and processed by specifying and manipulating patterns among objects in the database. Users can use the query language to specify a complex pattern of object class associations as search conditions for identifying those object instances of the corresponding classes that satisfy the pattern. This pattern is called a query graph, which is a subgraph of the schema graph. The following queries will be used throughout this work to illustrate the concept of pattern-based query processing.

Query 1. For all sections, find the majors of those research assistants who are taking these sections.

To satisfy this query, a pattern containing classes RA, Grad, Student, Section, and Department is specified as shown in Figure 3.3. In the query graph, a circle represents a class, and an edge represents that the instances of the two adjacent

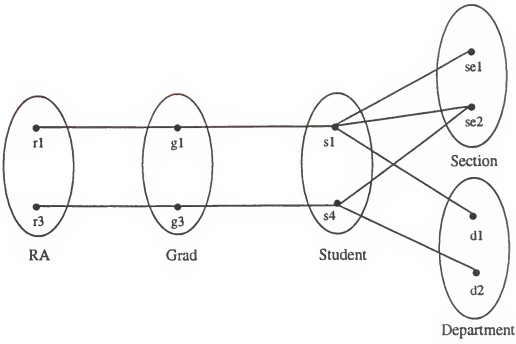


Figure 3.4 Resulting subdatabase of Query 1.

classes must be associated with each other. The AND branching condition states that an instance of class Student must be associated with instances in both classes Section and Department. The answer to this query contains those instances and their associations which satisfy the specified pattern. The resulting subdatabase is shown in Figure 3.4.

Suppose the university has a rule which states that a student cannot major and minor in the same department. To check this rule, the following query needs to be issued.

Query 2. List students who major and minor in the same department.

The query graph of this query is shown in Figure 3.5. According to the query, a student should associate with instances in both classes Undergrad and Department, and these two paths should merge at the same Department instance. These connections form a loop and imply three AND conditions as illustrated in the figure. The resulting subdatabase of this query is shown in Figure 3.6.

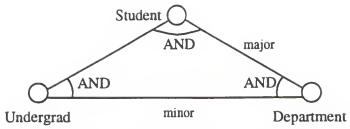


Figure 3.5 Query graph of Query 2.

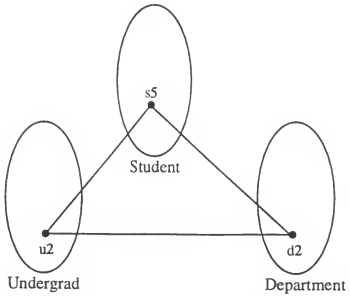


Figure 3.6 Resulting subdatabase of Query 2.

Since query patterns specified by the user can be very complex having linear, tree, or network structures and pattern searches have to be carried out in a potentially very large extensional database, the query processing is very costly and time-consuming, especially if patterns are combined together through multiple join operations [Kris86, Swam88, Ioan90, Schn90]. In order to avoid accessing and processing a large amount of data, a two-phase query processing strategy has been proposed [Lam89, Thak90b]. In the first phase, objects that satisfy the query pattern in the database are identified in the form of IIDs by processing and propagating IIDs among object classes. In the second phase, system- and/or user-defined functions are executed on the objects selected during the first phase to produce the final result. Since the retrieval of sizable descriptive data is postponed until the second phase when the relevant objects have been identified, this technique can avoid accessing and processing a large amount of unnecessary data in large OODB systems. Furthermore, in order to reduce the time and space for processing and storing intermediate results, objects that satisfy the query pattern in each object class are identified by searching the query pattern and marking object instances in the database instead of creating large intermediate results. This technique has another advantage in that the original structural properties of these instances are preserved and can be utilized by system- and/or user-defined operations during the second phase of query processing.

In a sequential algorithm for performing a pattern search, a traversal of each object instance through its association links would be necessary to verify if it satisfies a search condition. Often, after a long traversal of a path in an extensional database, it is found that the partially matched pattern does not satisfy the entire search pattern. In that case, the partially matched pattern needs to be taken out of the final result by deleting the object instances and association links that form the

pattern. For example, as shown in Figure 3.2 and Figure 3.3, if the query processor evaluates Query 1 starting from class RA, it will mark object instances r1, r2, r3 in RA, g1, g2, g3 in Grad, and s1, s3, s4 in Student when it reaches the class Student. After checking the connection information in Student, the query processor finds out that s3 is not connected to any object in Section and therefore the partially matched pattern r2—g2—s3 needs to be deleted through a backward traversal. This deletion of partially matched patterns is called the ripple back effect. Due to this effect, a sequential process cannot identify the correct answer with a single path traversal of the object graph. It is inefficient to trace and to delete partially matched patterns that have been previously selected. Moreover, in order to navigate an object graph efficiently, it is necessary to find the optimal traversal plan and this is not an easy task, either. Thus, efficient algorithms for pattern verification are needed to support query processing in OODBs.

3.3 A Graph Problem

The schema graph of an object-oriented database D is defined as $SG_D(C_D, A_D)$, where C_D is a set of vertices representing object classes; $A_D = \{(Ci, Cj)_k\}$ is a set of edges and each edge $(Ci, Cj)_k$ represents the kth association between classes Ci and Cj, where $Ci, Cj \in C_D$. The object graph of D is defined as $OG_D(O_D, E_D)$, where $O_D = \{Oi_p\}$ is a set of vertices, each of which, Oi_p, represents the pth object instance in class Ci; $E_D = \{(Oi_p, Oj_q)_k\}$ is a set of edges representing associations among object instances and each edge $(Oi_p, Oj_q)_k$ specifies that the pth instance in class Ci is related to the qth instance in class Cj through the kth association between classes Ci and Cj.

It is assumed that a query Q is issued against the database D. The query graph of Q is defined as an undirected graph $QG_Q(C_Q, A_Q)$, where C_Q is a set of vertices

representing the object classes referenced in Q and $A_Q = \{(Ci, Cj)_k\}$ is a set of edges representing the associations among the object classes referenced in Q. Also, it is assumed that all branches in QG_Q are AND branches in the examples to be used throughout this work. Algorithms for OR branches are similar but will not be presented here in order to avoid the duplication. The portion of D referenced by Q can be represented by an undirected object graph $OG_Q(O_Q, E_Q)$. In this graph, O_Q is a set of vertices representing the objects in the referenced classes, and $E_Q = \{(Oi_p, Oj_q)_k\}$, where $Oi_p \in Ci$, $Oj_q \in Cj$, and $(Ci, Cj)_k \in A_Q$, is a set of edges representing the links between the objects. Note that the object graph OG_Q is a subgraph of OG_D and the query graph QG_Q is a subgraph of SG_D. The problem is to identify all objects that satisfy the given query pattern.

The query processing problem is equivalent to a graph problem of finding the union of all subgraphs of OG_Q such that each subgraph can be mapped to the query graph QG_Q. For each such subgraph $OG_{sub}(O_{sub}, E_{sub})$, where $O_{sub} \subseteq O_Q$ and $E_{sub} \subseteq E_Q$, the vertex set O_{sub} contains the same number of vertices as the number of classes in C_Q with one from each class. In addition, for any two vertices in O_{sub}, Oi_p and Oj_q, $(Oi_p, Oj_q)_k \in E_{sub}$ if and only if $(Ci, Cj)_k \in A_Q$. Thus, the result of query Q is a union of some subgraphs of the object graph OG_Q and therefore the query processing problem can be transformed into a graph problem. It should be noted that only query graphs with AND branches are considered here, but the problem of processing query graphs with AND/OR branches can be defined in a similar fashion. Also, for the reason of notation simplicity, it is assumed in the subsequent chapters that at most one association is specified between any two classes so that the subscript k in $(Ci, Cj)_k$ can be omitted.

CHAPTER 4
IDENTIFICATION AND ELIMINATION APPROACHES

Identification and elimination are two general processing approaches that can be applied to solving the pattern-based query processing problem. The former approach has been introduced in Thakore [Thak90b] and Thakore et al. [Thak90a]. We will present a modification of this approach which serves as a contrast to a new elimination approach to be presented in this chapter. Both identification and elimination approaches for solving tree-structured queries are introduced together with a combined approach for solving queries with cyclic query graphs. Based on the formal graph model presented in Chapter 3, the correctness of each of the algorithms is proved in this chapter.

4.1 Solving Tree-structured Queries by Identification

The method of identifying tuples that satisfy some search condition has been commonly applied in relational databases. For example, semijoin operations are used to reduce the number of tuples involved in the evaluation of a query by identifying tuples whose attribute values satisfy the join predicate. The semijoin program evaluates a tree-structured query in two passes to find the tuples in all the relations that satisfy the query [Bern81]. It first performs semijoins in a breadth-first, leaf-to-root order in every edge of the query graph and then applies semijoins from the root down the tree toward the leaves. This identification-based approach can also be used in OODBs. The algorithm to be described below is applicable to tree-structured queries with AND branches.

Suppose $QG_Q(C_Q, A_Q)$ is the query graph of query Q and $OG_Q(O_Q, E_Q)$ is the undirected object graph of the part of database D that is referenced by query Q. Let $OG_{res}(O_{res}, E_{res})$, where $O_{res} \subseteq O_Q$ and $E_{res} \subseteq E_Q$, be the resulting graph of query Q (or the union of all subgraphs that are mapped to QG_Q as defined in Section 3.3). The following lemma shows that a query has a unique solution.

Lemma 1. For a Q and a D, there exists a unique OG_{res}.

Proof: Suppose we have two resulting graphs OG_{res_1} and OG_{res_2}. Since both OG_{res_1} and OG_{res_2} are the union of all the qualified subgraphs, $OG_{res\_1} \subseteq OG_{res\_2}$ and $OG_{res\_1} \supseteq OG_{res\_2}$. Then, $OG_{res\_1} = OG_{res\_2} = OG_{res}$, and OG_{res} is the unique solution. □

An identification-based algorithm selects vertices in the object graph that satisfy the query pattern. The object classes referenced by a query (i.e., those in a query graph) are classified into two types. Classes that have less than two edges in the query graph are called terminal classes. Otherwise, they are called non-terminal classes. The proposed identification process starts identifying object instances in terminal classes and then propagates messages which contain direct connection information to the associated classes. After the messages reach the associated class, the process proceeds with marking connected object instances if the instance has not been marked by a message from the same class before. If a new object instance is marked, messages are passed to other classes according to the following rules.

1. if Oi_p has been marked by messages from all its connected classes in the query but one, Cj, we will pass a message to class Cj to mark all its associated object instances that are associated with Oi_p;

2. if Oi_p has been marked by messages from all connected classes in the query, we will mark Oi_p as selected and pass messages to the neighboring classes in the query to mark all their object instances that are associated with Oi_p;

3. otherwise, we do not pass messages.

This identification process will continue until no more new object instance is marked. All vertices that are marked as selected and edges between them satisfy the query pattern. The algorithm is given as procedure MARK in Figure 4.1.

Theorem 1. For any Q and D, if QG_Q is acyclic, then procedure MARK can generate the resulting graph of query Q.

Proof: First, it is necessary to prove that procedure MARK will terminate. Assuming that there are m classes in QG_Q and n vertices in OG_Q. According to the above rules, each vertex will become idle after receiving at most $(m-1)$ tokens from its connected classes. Therefore, procedure MARK can generate the graph OG_{out} in $O(mn)$ steps of processing the received tokens.

The next step is to prove that procedure MARK generates a correct resulting graph for an acyclic query, i.e. $OG_{out} = OG_{res}$. As shown in Lemma 1, the resulting graph $OG_{res}(O_{res}, E_{res})$ is a unique solution to Q.

The output graph $OG_{out}(O_{out}, E_{out})$ of procedure MARK is a collection of "selected" vertices and edges. A vertex in OG_Q is selected because it has been marked by messages from all its connected classes. Therefore, any selected vertex Oi_p in OG_Q is connected to at least one vertex from each of Ci's connected classes in the query graph. According to procedure MARK, such a connected vertex Oj_q must satisfy one of the following conditions: (1) Cj is a terminal class; (2) Oj_q has been marked by messages from all connected classes except Ci; or (3) Oj_q is selected. If

```
PROCEDURE MARK(QG_Q(C_Q, A_Q), OG_Q(O_Q, E_Q))
  test single class;   /* if QG_Q has only one class then return OG_Q */
  initialization;      /* OG_out ← ∅; unmark all vertices in OG_Q */

  FOR every terminal class Ci in QG_Q DO
    FOR every vertex Oi_p of Ci in OG_Q DO
      FOR every connected vertex Oj_q in the connected class Cj DO
        pass token_i to Oj_q;
      END FOR;
    END FOR;
  END FOR;

  REPEAT
    FOR every vertex Oi_p in OG_Q DO
      IF Oi_p has received a token and not been marked with it before THEN
        mark Oi_p with the token;
        IF Oi_p has tokens from all connected classes except Cj THEN
          FOR every connected vertex Oj_q in Cj DO
            pass token_i to Oj_q;          /* rule 1 */
          END FOR;
        ELSE IF Oi_p has tokens from all connected classes THEN
          mark Oi_p with token_select;     /* rule 2 */
          FOR every connected vertex Oj_q in any connected class Cj THEN
            pass token_i to Oj_q;
          END FOR;
        END IF;
      END IF;
    END FOR;
  UNTIL no new vertex has been marked by a token;
```

Figure 4.1 Identification algorithm for tree-structured queries.

```
  FOR every vertex Oi_p in O_Q DO
    IF Oi_p is marked with token_select THEN
      put Oi_p in OG_out;
    END IF;
  END FOR;
  FOR every edge (Oi_p, Oj_q) in E_Q DO
    IF both Oi_p and Oj_q are marked with token_select THEN
      put (Oi_p, Oj_q) in OG_out;
    END IF;
  END FOR;

  RETURN OG_out;
END PROCEDURE;
```

Figure 4.1 (continued)

Oj_q is not selected under conditions (1) and (2), procedure MARK will mark Oj_q with *token_i* and eventually mark Oj_q as selected. Because both Oi_p and Oj_q are selected, edge (Oi_p, Oj_q) will be included in OG_{out} as well. Therefore, any vertex Oi_p in OG_{out} has at least one connected vertex in each connected class of Ci. In OG_{out}, we can start to traverse from Oi_p and find one connected vertex from each connected class to form a tree. The tree just created is a subgraph of OG_Q and can be mapped to the query graph as defined in Section 3.3. Thus, every vertex in OG_{out} is also in OG_{res}. Similarly, we can prove that every edge in OG_{out} is also an edge in OG_{res}.

It has been proved that $O_{out} \subseteq O_{res}$. Now suppose that vertex Oi_p is in OG_{res} but is not in OG_{out}. Since Oi_p is in OG_{res}, there exists a subgraph OG_{sub1} of OG_Q, which contains Oi_p and can be mapped to QG_Q as defined in Section 3.3. If we apply procedure MARK to OG_{sub1} alone, Oi_p will be marked as selected. This contradicts with the assumption, and therefore every vertex in OG_{res} will be in OG_{out}. Similarly, every edge in OG_{res} will also be in OG_{out}. This implies that

$O_{out} = O_{res}$ and $E_{out} = E_{res}$. Thus, it can be concluded that $OG_{out} = OG_{res}$ and procedure MARK can generate the correct result. □

4.2 Solving Tree-structured Queries by Elimination

Algorithms based on the elimination approach delete instances that do not satisfy the query pattern from the object graph. When an object instance is deleted from the object graph, all its edges are also deleted. This process may create new unqualified instances, causing the elimination process to be repeated until all the unqualified instances have been deleted. It is shown in this section that tree-structured queries can be solved using the elimination approach.

A qualified vertex for the type of query in consideration is defined as follows. Assume that $OG_{sub}(O_{sub}, E_{sub})$ is a subgraph of the object graph OG_Q. A vertex $Oi_p \in O_{sub}$ is a qualified vertex in OG_{sub} if, for any class Cx that is connected to Ci (i.e., $(Ci, Cx) \in A_Q$), there exists at least one instance Ox_z that is connected to Oi_p (i.e., $(Oi_p, Ox_z) \in E_{sub}$). In other words, the query graph serves as a pattern, and a vertex is qualified if it has at least one edge for each neighboring class in the pattern; otherwise it is unqualified.

The deletion of a vertex is defined as the deletion of the vertex and all its edges from the graph. The procedure DELETE for the deletion of unqualified vertices is shown in Figure 4.2. Before we prove that DELETE can generate the correct answer, the following lemma shows that the resulting graph is a subgraph of the output of DELETE.

Lemma 2. For any QG_Q and OG_Q, the resulting graph OG_{res} is a subgraph of the graph generated by procedure DELETE.

Proof: Since the object graph contains a limited number of objects, procedure DELETE will terminate after having deleted all unqualified vertices and their edges

PROCEDURE DELETE($QG_Q(C_Q, A_Q)$, $OG_Q(O_Q, E_Q)$)
    $OG_{out} \leftarrow OG_Q$;
    REPEAT
        delete unqualified vertices from OG_{out};
    UNTIL all vertices in OG_{out} are qualified;
    RETURN OG_{out};
END PROCEDURE;

Figure 4.2 Elimination algorithm for tree-structured queries.

in $O(mn)$ steps. The output of procedure DELETE, $OG_{out}(O_{out}, E_{out})$, is a subgraph of OG_Q and contains only qualified vertices. The unique resulting graph $OG_{res}(O_{res}, E_{res})$ is the union of all subgraphs that are mapped to QG_Q as defined in Section 3.3. According to the definition, every vertex in O_{res} must be a qualified vertex. Since procedure DELETE removes only unqualified vertices, any deleted vertex cannot be a member of O_{res}. In addition, any edge removed by procedure DELETE is connected to an unqualified vertex and cannot be a member of E_{res}. Therefore, $O_{res} \subseteq O_{out}$, $E_{res} \subseteq E_{out}$, and OG_{res} is a subgraph of OG_{out}. □

Lemma 2 says that OG_{res} is a subgraph of OG_{out} for any query graph QG_Q. The following theorem shows that if we restrict QG_Q to acyclic graph, then $OG_{out} = OG_{res}$.

Theorem 2. For any Q and D, if QG_Q is acyclic, then Q can be evaluated by deleting all unqualified instances.

Proof: Procedure DELETE generates a graph OG_{out} by deleting all unqualified vertices in OG. As shown in Lemma 1 and Lemma 2, $OG_{res}(O_{res}, E_{res})$ is a unique solution to Q and is a subgraph of $OG_{out}(O_{out}, E_{out})$. Suppose that vertex Oi_p is a member of O_{out} but is not a member of O_{res}. Since every vertex in O_{out} is qualified. we can start to traverse the query tree from Oi_p and find one vertex from each class in the query to form a tree. The tree just created is a subgraph of OG_Q and can

be mapped to the query graph as defined in Section 3.3. This contradicts with the assumption, and therefore every vertex of O_{out} is also a member of O_{res}. Due to this property and the fact that $O_{res} \subseteq O_{out}$, we know that $O_{out} = O_{res}$. Similarly, it can be shown that $E_{out} = E_{res}$. Thus, we can conclude that $OG_{out} = OG_{res}$. □

4.3 Cyclic Queries

Queries with cycles in their query graph are different from tree-structured queries. In relational databases, cyclic queries cannot be processed using semijoins [Bern81]. Cyclic queries in OODBs are also special and neither procedure MARK nor procedure DELETE can correctly generate resulting graphs for them. An example is used to illustrate this problem. Suppose that there are two undergraduate students s6, s7 in the database. As shown in Figure 4.3, student s6 majors in d1 and minors in d2, and student s7 majors in d2 and minors in d1. If we issue Query 2 "list students who major and minor in the same department" against this extensional database, neither student s6 nor s7 should be selected. However, because of the presence of cycle in the query graph and the propagation of only direct connection information, it should be clear that neither the identification approach nor the elimination approach can find the correct answer for this query. For example, although d1 is associated with instances in both classes Student and Undergrad and u3 is associated with instances in both classes Student and Department, it does not mean that d1 and u3 are in the same cycle. Therefore, in order to process a cyclic query, it is necessary to distinguish each individual cycle in the object graph.

Kambayashi [Kamb85] surveyed several methods for processing cyclic queries. These methods first convert a cyclic query graph into a tree and then apply tree-structured query processing procedure to find the result. For example, the method of attribute addition is used by Kambayashi et al. [Kamb82] to remove cycles. In order

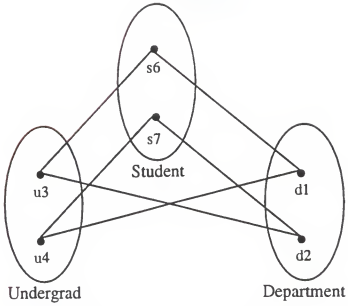


Figure 4.3 An example object graph.

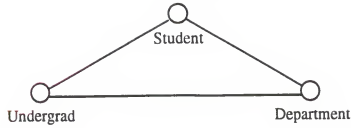
to convert cycles, a spanning tree of the given query graph is found. Since adding a non-tree edge back to the spanning tree can form a cycle with one non-tree edge, this method adds proper attributes to all relations in the cycle containing the non-tree edge. After obtaining a tree with additional attributes, generalized semijoins are applied in two passes, up-phase and down-phase, to find the final result. Another similar approach uses tagged semijoins to fully reduce cyclic databases [Tay89]. Rather than converting a cyclic query into a tree-structured query, this method performs all possible tagged semijoins until the database is fully reduced. Since the propagation of tagged attributes is not closely controlled by the algorithm, an extra number of tags may be used and some tagged attributes are unnecessarily copied to every relation. Therefore, as the author recognized, this approach may not be the best way for fully reducing a database state.

We propose a combined approach based on the identification and elimination techniques described to solve the cyclic queries. This strategy consists of the following two steps: (1) distinguishing individual cycles in the object graph and (2)

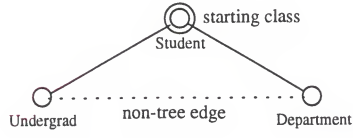
applying tree-structured query processing procedure to find the final result. To distinguish individual cycles in the object graph, we first find biconnected components in the given query graph. In a biconnected component, either there is only one edge or any two distinct edges that lie on a common cycle. We are interested in biconnected components containing more than one edge, or cycle, and we call these subgraphs cyclic components. Then, we find a spanning tree and select a class as the starting class for each cyclic component. Within the object graph of each cyclic component, unique tokens are propagated from each object instance in the starting class to all the reachable vertices and edges through the spanning tree. A vertex w (an edge (x,y)) is reachable from vertex v if there exists a path from vertex v to w (to (x,y)). After all the vertices and edges in the object graph corresponding to a spanning tree have been marked with tokens from the starting class, we need to mark edges in the object graph corresponding to non-tree edges of the spanning tree. For each non-tree edge (Ci, Cj) in a cyclic component, if both vertices Oi_p and Oj_q in classes Ci and Cj are marked with the same token, the edge (Oi_p, Oj_q) is marked with the same token. We use the example of evaluating Query 2 against the object graph shown in Figure 4.3 to illustrate the concept of cyclic query processing. The only cyclic component of the query graph and a spanning tree with starting class Student are shown in Figure 4.4(a) and Figure 4.4(b), respectively.

Figure 4.5 shows the object graph with tokens after we propagate tokens from the starting class to all the reachable vertices and edges through the spanning tree. Note that edges (u3, d2) and (u4, d1) are not marked because they are instances of a non-tree edge and do not have the same tokens on both vertices.

After distinguishing individual cycles for the query, we use the elimination approach to find the final result. As shown in Lemma 2, the output of deleting all the unqualified vertices from the object graph is a superset of the final result. Therefore,



(a) A cyclic component



(b) A spanning tree

Figure 4.4 Query 2

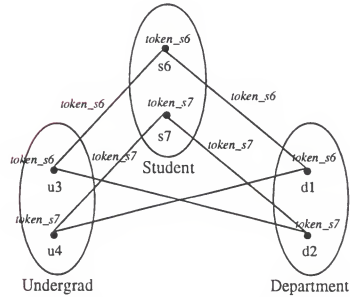


Figure 4.5 Cyclic query processing

we can apply procedure DELETE to remove unqualified vertices, regardless of the existence of cycles in the query graph. However, in order to find the final result, we define a qualified vertex and a qualified edge for a cycle as follows. For a set of vertices and edges which are marked with the same token in a cyclic component QG_{CC}, a vertex Oi_p is qualified for the token in QG_{CC} if, for any class Cx in QG_{CC} that is connected to Ci, there exists at least one vertex Ox_z that is connected to Oi_p and (Oi_p, Ox_z) is marked with the same token; an edge (Oi_p, Oj_q) is qualified for the token in QG_{CC} if Oi_p, Oj_q, and (Oi_p, Oj_q) are all marked with the same token. In other words, the marked subgraph can be mapped to the cyclic component as defined in Section 3.3. We need to delete unqualified vertices for the query graph and vertices and edges that are not qualified for any individual cycle in a cyclic component. For example, edges (u3, d2) and (u4, d1) shown in Figure 4.5 are not qualified for the cycle and need to be deleted from the object graph. Consequently, all vertices and edges in the object graph will be correctly deleted. The algorithm is given as procedure IEC in Figure 4.6.

Theorem 3. For any Q and D, if QG_Q is cyclic, then procedure IEC can generate the resulting graph of query Q.

Proof: Since tasks in IEC, such as finding biconnected components [Aho74], finding a spanning tree, identifying individual cycles in each cyclic component, and eliminating limited number of unqualified vertices, edges, and tokens, can be done in a finite time, the termination of procedure IEC is guaranteed.

We only delete unqualified vertices for the query graph and vertices and edges that are not qualified for any individual cycle in a cyclic component. Therefore, OG_{out} is a superset of OG_{res}. In OG_{out}, all vertices and edges with the same token are qualified for a cyclic component and they form a union of subgraphs that can

```
PROCEDURE IEC(QG_Q(C_Q, A_Q), OG_Q(O_Q, E_Q))
  find all cyclic components in QG_Q;
  FOR every cyclic component QG_CC DO
    find a spanning tree of QG_CC;
    select Ci as the starting class;
    FOR every Oi_p in Ci DO
      mark all vertices and edges reachable from Oi_p through the pattern
      of the spanning tree with token_ip;
    END FOR;
    FOR every non-tree edge (Cx, Cy) in QG_CC DO
      FOR every edge (Ox_p, Oy_q) DO
        IF both Ox_p and Oy_q are marked with the same token THEN
          mark (Ox_p, Oy_q) with the token;
        END IF;
      END FOR;
    END FOR;
  END FOR;

  OG_out ← OG_Q;
  REPEAT
    IF a vertex is unqualified for QG_Q THAN
      delete the vertex and its edges from OG_out;
    END IF;
    IF a vertex Oi_p is unqualified for a token (say token_xz) in a QG_CC THEN
      delete token_xz from Oi_p and from all its edges;
      IF Oi_p does not have any token in QG_CC left THAN
        delete Oi_p and its edges from OG_out;
      END IF;
    END IF;
    IF (Oi_p, Oj_q) is unqualified for a token (say token_xz) in a QG_CC THEN
      delete token_xz from (Oi_p, Oj_q);
      IF (Oi_p, Oj_q) does not have any token in QG_CC left THAN
        delete (Oi_p, Oj_q) from OG_out;
      END IF;
    END IF;
  UNTIL no new token, vertex or edge is deleted from OG_out;
  RETURN OG_out;
END PROCEDURE;
```

Figure 4.6 Algorithm for cyclic queries.

be mapped to the cyclic component as defined in Section 3.3. According to the definition, no cycle can be formed between biconnected components. If we treat each cyclic component as a pseudo vertex, a cyclic query graph can be seen as a tree-structured query graph and be processed by procedure DELETE. Since we can find subgraphs of OG_{out} from both outside and inside the cyclic components that can be mapped to the query graph, procedure IEC does correctly generate the resulting graph. □

CHAPTER 5
IDENTIFICATION- AND ELIMINATION-BASED PARALLEL ALGORITHMS

We shall present parallel algorithms based on the identification and elimination approaches for tree-structured and cyclic queries in this chapter. Both parallel algorithms are multiple wavefront algorithms and can be implemented on a shared-nothing architecture. We also discuss the issues of starting the algorithms from less number of classes and the implementation of inter-class attribute comparisons.

5.1 Architecture and Data Organization

The multiple wavefront algorithms are supported by an asynchronous parallel processing system which consists of a set of processing nodes interconnected by an interconnection network. Each processing node contains a processing unit, main memory, and secondary storage devices. Processing nodes do not share memory or a global clock; they can only communicate by sending and receiving messages. The communication channels are bi-directional. It is assumed that no messages are lost and that messages are always received in the order in which they are sent.

Different interconnection networks can be used to implement the proposed parallel algorithms. Multiple-processor systems with a high degree of connectivity per node (e.g., nCUBE and Hypercube) would be ideal since the schema graph can be mapped to the processing nodes in a fashion so that connectivity among classes will match as closely as possible to the physical connections of the processing nodes and communication and data transfer time among processing nodes can be reduced.

It is important to have an effective strategy for distributing data across multiple nodes in a shared-nothing multiprocessor environment. Since the strategies

used for clustering, partitioning and mapping data can be applied to any wavefront algorithms presented in this chapter, we shall use a simple data organization and mapping method. In order to localize retrieval, manipulation, and user-defined operations and to reduce the overall communication among processors, all the data associated with an object class are grouped together. We use a simple class-per-processor mapping strategy and assume that the interconnection network provides direct channels between associated object classes. However, multiple classes can be assigned to a single physical processor and two processors may need to communicate through some other processors. These issues will be studied in the subsequent chapters.

The associations of instances between two associated object classes can be represented by an adjacency matrix. Rows and columns of the adjacency matrix are stored as adjacency lists of IIDs in the processors that manage the two classes. The adjacency list for each instance contains IIDs of the connected objects in the associated class. For supporting the parallel algorithms based on both identification and elimination approaches, the set of adjacency lists are stored in a data structure called "backward pointer". Instead of storing connected IIDs of each instance in the class, we store the list of local IIDs that are connected to instances in another class. Also, an integer array is used to register the number of connections from each local instance to instances in another class. For example, the object graph of the part of the university database referenced by Query 1 is shown in Figure 5.1 and the data structure representing this object graph is shown in Table 5.1. We assume that the data and methods defined in the five object classes: RA, Grad, Student, Section, and Department are stored in processors P1, P2, P3, P4, and P5, respectively. The adjacency lists for Grad instances that are connected to RA instances, g1 $\rightarrow$ r1; g2 $\rightarrow$ r2; g3 $\rightarrow$ r3, are stored in processor P1 to represent the

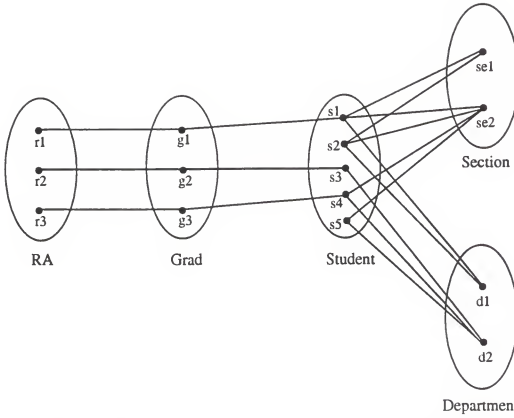


Figure 5.1 Object graph of the subdatabase referenced by Query 1.

Table 5.1 Data structure representing the object graph referenced by Query 1.

Processor	Class	To Class	CON	Backward Pointer
P1	RA	Grad	r1 1 r2 1 r3 1	g1 → r1 g2 → r2 g3 → r3
P2	Grad	RA	g1 1 g2 1 g3 1	r1 → g1 r2 → g2 r3 → g3
		Student	g1 1 g2 1 g3 1	s1 → g1 s2 → s3 → g2 s4 → g3 s5 →
P3	Student	Grad	s1 1 s2 0 s3 1 s4 1 s5 0	g1 → s1 g2 → s3 g3 → s4
		Section	s1 2 s2 2 s3 0 s4 1 s5 1	se1 → s1, s2 se2 → s1, s2, s4, s5
		Department	s1 1 s2 1 s3 1 s4 1 s5 1	d1 → s1, s2 d2 → s3, s4, s5
P4	Section	Student	se1 2 se2 4	s1 → se1, se2 s2 → se1, se2 s3 → s4 → se2 s5 → se2
P5	Department	Student	d1 2 d2 3	s1 → d1 s2 → d1 s3 → d2 s4 → d2 s5 → d2

connection information of class RA. Two sets of adjacency lists, r1 → g1; r2 → g2; r3 → g3 and s1 → g1; s2 → ; s3 → g2; s4 → g3; s5 → , are stored in processor P2 to explicitly specify the associations of Grad instances with RA and Student instances, respectively. Note that the explicitly stored adjacency lists for each object instance can be viewed as precomputed joins in relational databases [Vald87, Bic89].

For each class association, one integer array CON is associated with each object class as shown in Table 5.1. Each element of the array is corresponding to one instance of the class stored in the processor, and the integer value is the number of connections between that instance and the instances of the associated class. For example, the elements of array Section.CON in class Student, which represent the connections to class Section, have values 2, 2, 0, 1, and 1 for instances s1, s2, s3, s4, and s5, respectively. Note that this CON array can be created locally by reading the reversely stored adjacency lists.

We also vertically partition the descriptive data of each object class and store them seperately as shown in Table 5.2. This vertical partition scheme allows individual domain class to be retrieved. Therefore, if only one of several domain classes of an object class is referenced in a query, the retrieval of unnecessary data can be avoided. However, this scheme needs more space to store IIDs because they are duplicated in each partition. Also, if all descriptive data of an object are needed, we must retrieve corresponding data from all partitions of that class. This partition scheme is similar to other vertical partition schemes proposed for relational systems [Cope85, Khos88, Vald87].

Table 5.2 Data structure representing descriptive data.

Processor	Class	To Class	IID	Value
P3	Student	GPA	s1	3.5
			s2	3.2
			s3	3.9
			s4	3.8
			s5	3.0
P4	Section	SectionNum	se1	3729
			se2	2613
		Room	se1	E121
			se2	E210
		TextBook	se1	Database Computers
			se2	Transaction Processing
P5	Department	Name	d1	EE
			d2	CIS
		College	d1	Engineering
			d2	Engineering

5.2 Identification-based Parallel Algorithm for Tree-structured Queries

We modify the MARK procedure to make it suitable for the shared-nothing environment. The resulting algorithm is an identification-based parallel multiple wavefront algorithm, which is applicable to tree-structured queries with AND branches. A slightly different algorithm is used for query structures with OR branches.

The selection and propagation of IIDs are dynamically determined in each class by the following rules. Suppose a class C is stored in processor P,

- if C is a terminal class, processor P will start the IID propagation process. If processor P receives the stream of IIDs from its only neighbor, it will mark those local objects that are connected by the received IIDs as selected objects and then terminate.

- if C is a non-terminal class,

1. if processor P has received streams of IIDs from all its neighbors but one, it will process the received IIDs and select those C instances that satisfy the selection condition and the query pattern. Then, it will send the IIDs of class C instance that have associated instances in the only remaining neighboring class to the corresponding processor;
2. if processor P receives the last incoming streams of IIDs, it will form the final result of the query for class C and then pass the IIDs of those class C instances to associated instances of all neighbors except the sender of the last stream.
3. otherwise, processor P processes the received streams of IIDs but does not send out any message;

Each propagation of IIDs forms a wavefront. Multiple wavefronts will start simultaneously at the terminal nodes and propagate IIDs according to the above rules. Because it is assumed that all branches in the query graph are AND branches, a processor can not propagate IIDs until it has received and treated all but one incoming streams of IIDs. Different from the semijoin approach reported [Bern81] and the active graph model [Bic89], this parallel algorithm is not restricted to a single root node. Rather, the identification process starts from multiple nodes, thus rendering more flexibility and efficiency.

Using the example query graph of Query 1 and database shown in Figure 5.1 and Table 5.1, the execution of the parallel identification algorithm is shown in Figure 5.2. Assuming that the propagation of IIDs in each step will take one time unit, the selected instances in all the classes and the IIDs in transit are shown in Figure 5.2 for each elapsed time unit. All processors that handle terminal classes initiate the wavefronts. In this example, processor P1 finds out that Grad.CON[r1] = 1,

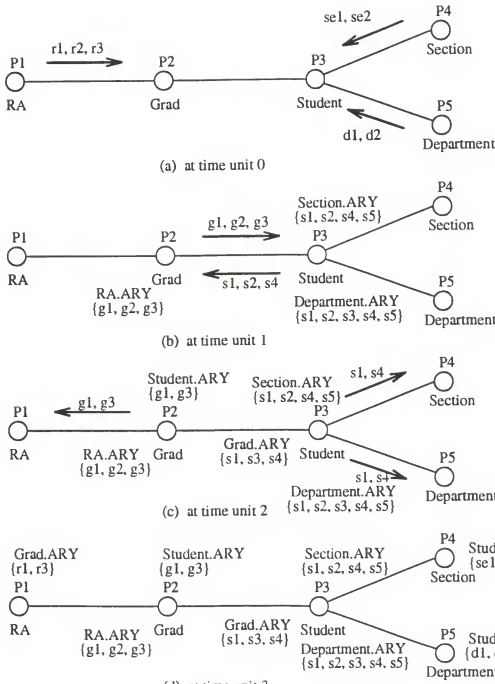


Figure 5.2 Execution of the identification algorithm.

Grad.CON[r2] = 1, and Grad.CON[r3] = 1 in class RA. This means that each of r1, r2, and r3 is connected to one instance in Grad. Therefore, processor P1 sends a messages which contains IIDs of r1, r2, and r3 to processor P2. After receiving the message from RA, processor P2 (or class Grad) checks the backward pointers and marks instances g1, g2, and g3 because they are connected to r1, r2, and r3. According to the propagation rules, class Grad receives all but one wavefront from its neighbors and therefore it can propagate the message to class Student. This process continues until the wavefront reaches terminal classes. It should be obvious that this parallel algorithm avoids the problem of sequential ripple back effect and produces the correct resulting subdatabase as shown in Figure 3.4.

5.3 Elimination-based Parallel Algorithm for Tree-structured Queries

In this algorithm, we use arrays (ARYs) to store temporary data and the output. Each element of the array is corresponding to one instance of the class stored in the processor, and the integer value is the number of remaining connections between that instance and the instances of the associated class. Initially, these arrays have identical numbers of the CON arrays as shown in Table 5.1. For example, the elements of array Section.ARY in class Student, which represent the connections to class Section, have initial values 2, 2. 0, 1, and 1 for instances s1, s2, s3, s4, and s5, respectively. The number of connections is reduced by one every time a connection is deleted during the execution of the algorithm. When it becomes zero, the corresponding instance is then deleted.

Since the connection information is stored with each object, the process of checking and deleting object instances and their associations can be done independently in every processor which contains those instances. Whenever one processor

deletes an object instance that has connections with object instances in other processors, the processor propagates the connection information to the destination processors, and then the processors that receive the messages will start their own checking and deleting procedures. This propagation process will continue until no more elimination can be found. An active processor checks its ARY arrays for deleting instances, and then propagates the elimination information, if any exists, to its neighboring classes. An object instance needs to be deleted if it has corresponding elements with zero values in some ARY arrays and with non-zero values in other ARY arrays (this is only true for a query with AND branch(es)). When the ARY array element of a deleted instance has a non-zero value, the ARY array element is assigned to 0 and the deleted instance's IID is sent to the associated class (i.e., its processor). After a processor finishes all the elimination and propagation tasks, it becomes idle again. For example, as shown in Table 5.1 and Figure 5.3(a), instances s2, s3, and s5 need to be deleted in class Student because Grad.CON[s2] = 0, Section.CON[s3] = 0, and Grad.CON[s5] = 0. After assigning 0 to Section.ARY[s2], Department.ARY[s2], Grad.ARY[s3], Department.ARY[s3], Section.ARY[s5], and Department.ARY[s5], processor P3 sends elimination messages s2 and s5 to processors P4 and P5, and s3 to processors P2 and P5. It then becomes idle and waits for new messages.

When a processor receives a list of IIDs from a neighboring processor, it obtains the adjacency list for each received IID from the set of adjacency lists that corresponds to the sending processor. For each local IID in the adjacency list, the processor decreases the value of the corresponding ARY element by 1. After processing all the received IIDs, the processor starts the checking process described above. For example, as shown in Figure 5.3(b), after receiving IID s3 from processor P3, P2 finds that s3 connects to g2 and then it decreases Student.ARY[g2] from 1 to

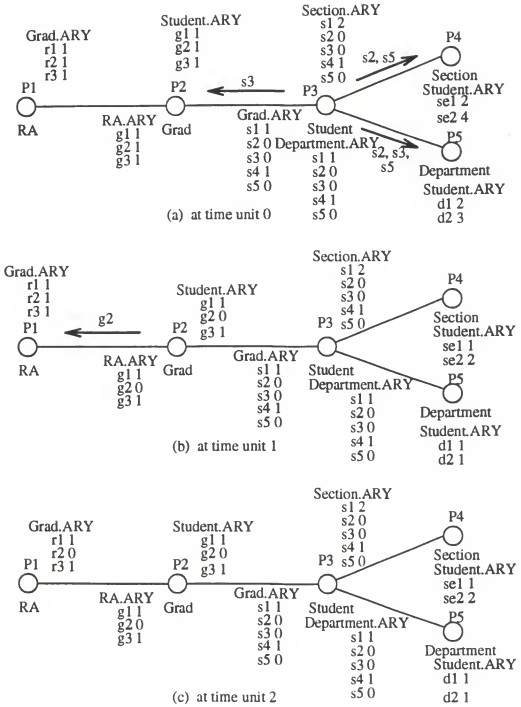


Figure 5.3 Execution of the elimination algorithm.

0. At this time, P2 finds out that g2 is an unqualified object. It assigns a 0 to RA.ARY[g2] and sends an elimination message g2 to P1. Thus, the deletion wavefront will continue to propagate until it cannot find any more unqualified instances. Note that the processors can process the received IIDs in a pipelined fashion and thus increase the degree of parallelism.

An asynchronous parallel algorithm terminates if and only if all processors are idle and no message is in transit. Because only tree-structured queries are considered here, each processor can determine the completion of the algorithm by simply counting the number of incoming end tokens, which are issued by processors that start the wavefronts. In the parallel elimination algorithm, all processors start the deletion wavefronts and then propagate end tokens to all its neighboring processors. Therefore, each processor executing the elimination algorithm can complete its computation after receiving the same number of end tokens as the number of processors participated in the query.

5.4 A Combined Parallel Algorithm for Cyclic Queries

The IEC procedure introduced in Section 4.3 has two major tasks: identifying individual cycles and eliminating unqualified instances. In a shared-nothing environment, these two tasks can be executed partly in parallel. We can proceed the deletion of instances that are unqualified for the query pattern as defined in Section 4.2 without waiting for the results of the other task. However, the deletion of instances that are not qualified for each individual cycle must be postponed until the cycle has been identified.

We use an example to illustrate the execution of the combined parallel algorithm for cyclic queries. The object graph of the part of the university database referenced by Query 2 is shown in Figure 5.4. The data structure representing this

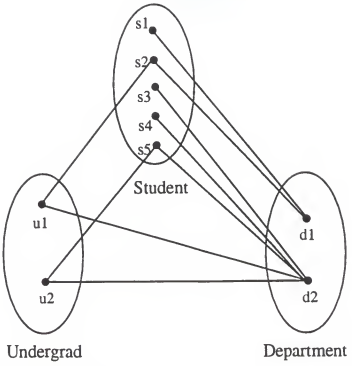
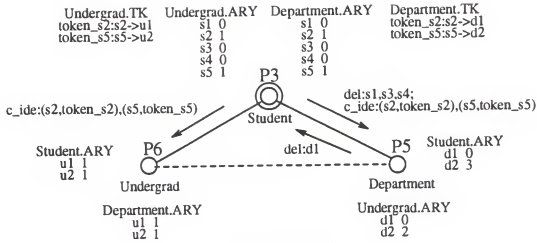


Figure 5.4 Object graph of the subdatabase referenced by Query 2.

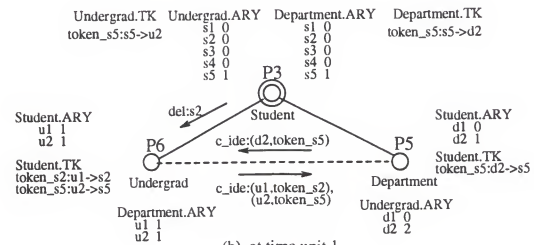
object graph is shown in Table 5.3. We assume that classes Student, Department, and Undergrade are assigned to processors P3, P5, and P6, respectively. We use ARYs to store the temporary data and output as in the elimination-based parallel algorithm introduced in the previous section. In addition, another structure, TK, is used to store the tokens propagated to each connection between instances. The execution of the combined algorithm for Query 2 is shown in Figure 5.5. There is only one cyclic component in the query graph. We select class Student as the starting class and the association between classes Undergrad and Department as the non-tree edge, which is shown as a dotted line in Figure 5.5. By checking CON arrays, instances s1, s3, and s4 in class Student and instance d1 in class Department are not qualified for the query pattern. Therefore, processor P3 assigns 0's to Department.ARY[s1], Department.ARY[s3], and Department.ARY[s4] and sends deletion message "del:s1,s3,s4" to processor P5; processor P5 assigns 0

Table 5.3 Data structure representing the object graph referenced by Query 2.

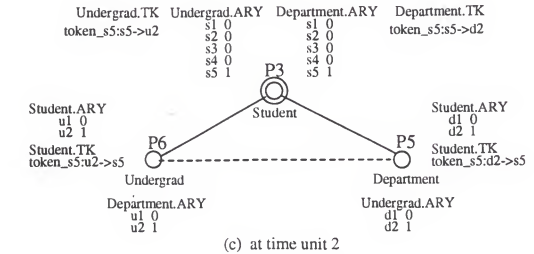
Processor	Class	To Class	CON	Backward Pointer
P3	Student	Department	s1 1	d1 → s1, s2
			s2 1	d2 → s3, s4, s5
			s3 1	
			s4 1	
			s5 1	
		Undergrad	s1 0	u1 → s2
			s2 1	u2 → s5
			s3 0	
			s4 0	
			s5 1	
P5	Department	Student	d1 2	s1 → d1
			d2 3	s2 → d1
				s3 → d2
				s4 → d2
				s5 → d2
		Undergrad	d1 0	u1 → d2
			d2 2	u2 → d2
P6	Undergrad	Student	u1 1	s1 →
			u2 1	s2 → u1
				s3 →
				s4 →
				s5 → u2
		Department	u1 1	d1 →
			u2 1	d2 → u1, u2



(a) at time unit 0



(b) at time unit 1



(c) at time unit 2

Figure 5.5 Execution of the combined algorithm.

to Student.ARY[d1] and sends deletion message "del:d1" to processor P3. In the meantime, processor P3 initiates the propagation of tokens from class Student by marking edge (s2, u1) with token_s2 and edge (s5, u2) with token_s5 and stores the information in Undergrad.TK. It also marks edges (s2, d1) and (s5, d2) with token_s2 and token_s5 respectively and stores them in Department.TK. Then, processor P3 propagates the information "c_ide:(s2,token_s2),(s5,token_s5)" to processor P6 and "c_ide:(s2,token_s2),(s5,token_s5)" to processor P5. All these changes and activities are illustrated in Figure 5.5(a).

After receiving the deletion message "del:d1", processor P3 checks the backward pointer d1 → s1, s2 and then decreases the value of Undergrad.ARY[s2] by 1. Since Undergrad.ARY[s2] becomes 0, processor P3 deletes s2's edges in TKs, and sends deletion message "del:s2" to processor P6. Processors P5 and P6 process the received c_ide information and place tokens to the corresponding edges in the structure of TK. Because classes Undergrad and Department are leaf nodes in the spanning tree, the propagation of tokens will not continue and they just exchange token information for identifying those non-tree edges that satisfy each individual cycle. The messages in transit and results of theses operations are shown in Figure 5.5(b). Figure 5.5(c) shows that the non-tree edge (u1, d2) is not qualified for any cycle and is deleted from the structures in processor P5 and P6 accordingly. We stop our illustration at this point because no more instances or edges will be deleted from the structures. However, we have to point out that end tokens need to be passed to all classes in the spanning tree in order for each processor to terminate the execution of this algorithm. These messages are not shown in the figures because they are already packed.

5.5 Less Degree of Parallelism

We discussed the identification- and elimination-based parallel algorithms for tree-structured queries in the previous sections. They both are multiple wavefront algorithms which start the wavefronts from a fixed number of classes. For example, the identification-based parallel algorithm starts from all terminal classes, whereas the elimination-based parallel algorithm starts from all classes in the query. Intuitively, the elimination algorithm would perform better because of the higher degree of parallelism. However, in some cases starting from all possible classes does not necessarily give the best performance. We present two cases in which activating a small number of processors may be beneficial: (1) If some of the processors involved in the query processing are busy with some other tasks (e.g., with other queries), we should start the algorithm from the less busy processors instead of giving more load to the busy processors; (2) If processing certain classes first can drastically reduce the amount of data that need to be transferred among processors, these classes should be processed first. We shall study the possibilities of starting the wavefront algorithms from less number of classes in this section.

Before discussing the details of each case, we first review the rules of propagation of identification wavefronts introduced in Section 5.2. The wavefronts starts from all terminal classes. In a non-terminal class, the wavefront will not be propagated until the class has received all but one wavefronts from its neighboring classes. As illustrated in Figure 5.6, class A with i associated classes will propagate the wavefront to the ith neighboring class B after it has received the $(i - 1)$ wavefronts. This means that every instance selected in class A satisfies a partial query pattern which includes class A and all classes to its left side. Therefore, when class A receives a wavefront from class B, the intersection of the new message with the previous results in A produces the final results of class A. Then, we propagate the

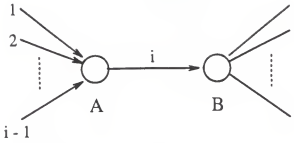


Figure 5.6 Propagation of wavefronts.

information to all (i – 1) neighboring classes to class A's left side so that they can form their final results.

If the processor that handles class B is busy with some other tasks when class A is ready to propagate the wavefront to class B, it may be beneficial or necessary to withhold the propagation until the processor of class B carries a lighter load. In the meantime, instead of waiting for the processor of class B without doing anything, class A can propagate a wavefront back to all classes to its left side to form a partial result which satisfies the partial query pattern. By doing this, we effectively reduce the size of database for processing when class B eventually propagates the wavefront back to class A.

We can apply this approach to the general case. If some of the processors involved in a query pattern are busy, the query pattern is partitioned into sub-patterns because the busy processors will block the propagation of wavefronts. For example, as shown in Figure 5.7, processor P2 which handles class B is busy and therefore the query pattern is partitioned into two sub-patterns. One sub-pattern contains class A and the other one contains classes C, D, and E. Each sub-pattern will be processed independently. When processor P2 is available, the wavefronts will start from classes A and C instead of from classes A, D, and E. We can use class C to represent the sub-pattern of C, D, and E because every instance selected in

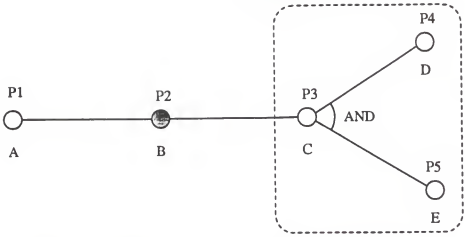


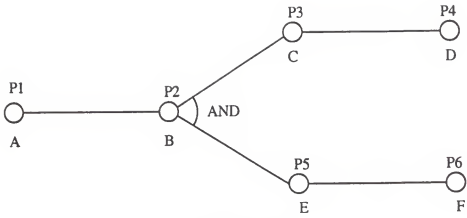
Figure 5.7 Pattern partition because of a busy processor.

class C satisfies the sub-pattern. We note that this method of partitioning the query pattern into sub-patterns is also applicable to the elimination-based algorithm.

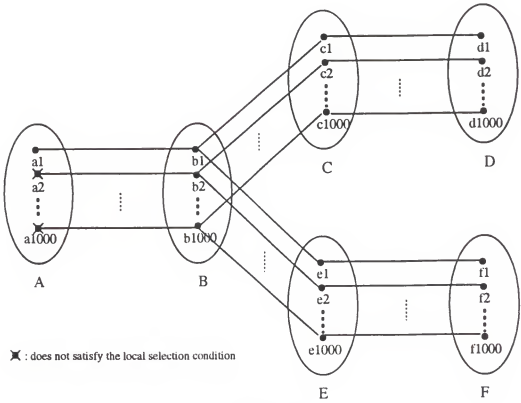
In the second case, starting the wavefronts from less number of classes may be more beneficial. For example, a query pattern and its corresponding object graph are shown in Figure 5.8(a) and (b), respectively. We assume that only one instance in class A satisfies the local selection condition specified in the query, while all instances in other classes satisfy the local selection conditions. If we start the identification wavefront algorithm from all terminal classes A, D, and F, the processing time can be approximately calculated as the following.

$$
\begin{aligned}
T &= 1000T_P + (1000T_C + 1T_O) + 1000T_P + (1000T_C + 1T_O) + 1000T_P + \\
&\quad (1T_C + 1T_O) + 1T_P + (1T_C + 1T_O) + 1T_P \\
&= 3002T_P + 2002T_C + 4T_O
\end{aligned}
$$

where T_P is the time for processing one instance, T_C is the communication time for propagating one instance IID, and T_O is the communication overhead. The first line of the above equation represents the time for a wavefront which starts from classes



(a) Query graph



(b) Object graph

Figure 5.8 An example.

D or F to reach class B. The second line represents the time for the final wavefront to reach classes D and F.

However, we can start an initial wavefront from class A alone to identify all connected instances in every class. After this initial wavefront reaches all terminal classes, we start the identification wavefront algorithm from all terminal classes except class A to work on those instances that are already identified by the first wavefront. In this example, the processing time of this method is approximately calculated as the following.

$$
\begin{aligned}
T &= 1000T_P + (1T_C + 1T_O) + 1T_P + (1T_C + 1T_O) + 1T_P + (1T_C + 1T_O) + 1T_P + \\
&\quad 1T_P + (1T_C + 1T_O) + 1T_P + (1T_C + 1T_O) + 1T_P + (1T_C + 1T_O) + 1T_P + \\
&\quad (1T_C + 1T_O) + 1T_P \\
&= 1008T_P + 7T_C + 7T_O
\end{aligned}
$$

The first line of this equation represents the time for the first wavefront to travel from class A to classes D and F. The second and third lines represent the time for the identification wavefront algorithm to work on the data identified by the first wavefront. Thus, if the communication overhead (T_O) is not very large, starting from class A alone in this example would take much less time to process the query. The reason that this method is better is because a very small number of instances are selected and processed in each class if we start the wavefront from class A. Although this method takes an extra wavefront propagation in the beginning, the overall execution time is still much less than the identification parallel algorithm which starts from all terminal classes.

In general, we can start multiple initial wavefronts from terminal classes as well as non-terminal classes. The purpose of these initial wavefronts is to reduce the

database size for later operations. Each wavefront will propagate from its starting class to all terminal classes in the query and identify all instances that are connected to instances in the starting class. If any instance is not identified by the wavefront, it will not be a qualified instance for the query. Since each initial wavefront just tries to find all the reachable instances from the starting class, they do not interfere with each other and can be run independently. After all the initial wavefronts reach terminal classes, only those instances that identified by every wavefront are still under consideration for the query. In the example, we use the identification algorithm to find the final results. Actually, we can use either identification or elimination algorithm to work on those identified instances. Also, we can start an initial wavefront from a chosen initial class and start sending elimination wavefronts from all other classes toward the initial class at the same time. Since the initial wavefront is a special elimination process which eliminates those instances that are not reachable from the initial class, it is not necessary to send elimination wavefronts away from the initial class.

5.6 Inter-class Comparison

In addition to the query pattern, inter-class comparison conditions can be specified in a query to further restrict the resulting subdatabase. These conditions are comparisons between some attributes of two classes and/or comparisons between objects (i.e., Where conditions in OQL [Alas89]). We will study the necessary messages and techniques for implementing these comparisons.

In a shared-nothing environment, processors can only communicate via message passing. Various information must be conveyed in different types of messages to accomplish the computation task. The parallel algorithms for tree-structured queries

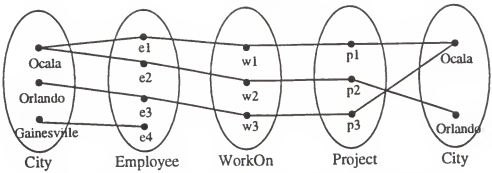


Figure 5.9 The object graph of an example database.

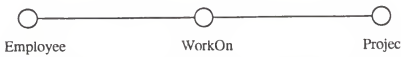
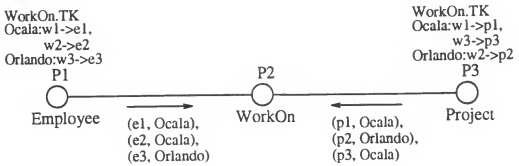


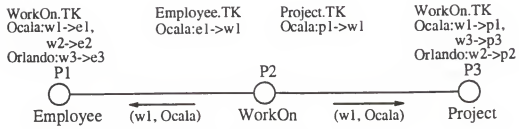
Figure 5.10 The query graph of the example of inter-class comparison.

use messages that contain only IIDs to pass the direct connection information between processors. The information in the messages is sufficient for identifying the subdatabase that satisfies the query pattern. The combined parallel algorithm for cyclic queries uses three different types of messages for deleting object instances, identifying individual cycles, and deleting unqualified object instances and connections for each cycle. For example, IID and token pairs are propagated from a starting class to all reachable vertices and edges through a spanning tree in a cyclic component to identify individual cycles. We will use similar messages to implement the inter-class comparisons.

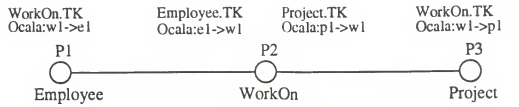
An example database may be used to illustrate the implementation of inter-class comparisons. Its object graph and query graph are shown in Figure 5.9 and Figure 5.10, respectively. Suppose there is a condition, Where Employee.City = Project.City, specified in the query. When we propagate wavefronts among the classes, we will use the city names as tokens to mark every reachable instances and edges. The execution of this example is shown in Figure 5.11. It is clear that instance



(a) at time unit 0



(b) at time unit 1



(c) at time unit 2

Figure 5.11 The execution of inter-class comparison.

w2 and w3 and their edges are deleted from the resulting subdatabase because they are marked with different cities by the two identification wavefronts. Due to the deletion of w2 and w3, instances e2 and e3 in class Employee and instances p2 and p3 in class Project are eventually deleted from the resulting subdatabase.

On the other hand, some retrieval operations may require that object instances of two classes be identified based on some relationship between their attribute values rather than the associations of these instances. For example, one may be interested in finding all employees and projects that are located in the same city rather than based on employees' associations with projects. In this case, the attribute values and IIDs of these two classes need to be exchanged. In order to exchange city names and IIDs between classes Employee and Project, a new type of message which contains pairs of city name and IID in one class will be propagated to the other class. The processor that manages the destination class will read the message and use the information to mark or process object instances that satisfy the attribute comparison condition.

CHAPTER 6
PARTITIONING A DATABASE FOR PARALLELISM

An effective strategy for distributing data across multiple disks in a shared-nothing multiprocessor environment is crucial to achieving good performance in a parallel object-oriented database management system. During query processing, a large amount of data need to be processed and transferred among the processing nodes in the system. A good data placement strategy should be able to reduce the communication overheads, and, at the same time, to provide the opportunity for exploiting different types of parallelism in query processing, such as intra-operator parallelism, inter-operator parallelism, and inter-query parallelism. However, there exists a conflict between these two requirements. While minimizing interprocessor communication favors the assignment of the whole database to a small number of processors, achieving higher degree of parallelism favors the distributions of the database evenly among a large number of processors. A trade-off must be made to obtain a good policy for mapping the database to the processors.

A simple file assignment problem, which deals with assigning files to different nodes of a computer network, has been studied extensively [Dowd82]. However, most of the works assume that a request is made at one site and all the data for answering it is transferred to that site. This simple view of application cannot model the query processing strategy in a parallel database system. The simple file assignment problem is an NP-complete problem. We need good heuristics to solve this and more complicated database allocation problems. In this chapter, we propose

some heuristics for partitioning an OODB so that the overall execution time can be reduced.

6.1 The Problem

The problem is to partition a given OODB and assign the partitions to the nodes in a multiprocessor system. It is assumed that the number of object classes in the database is larger than the number of processors in the system. Also, we assume that the processors are fully connected. This simplifies the problem so that we do not need to consider the effects of the network's physical topology. However, we can simulate different topologies by introducing various delays to different links.

We assume that the unit of distribution is class. In other words, classes are not allowed to be split, and each class must reside in one and only one node. Since we group all the data associated with an object class together, we can localize retrieval, manipulation, and user-defined operations and reduce the overall communication among processors. If we horizontally partition the classes and assign them to multiple processors, two sets of processors need to communicate with each other when two classes want to exchange information in our multi-wavefront algorithms. In addition, if a large number of processors work on the same class, this horizontal partition scheme does not provide a good environment for multiple queries to be executed in parallel when these queries access different classes. Thus, we choose class as the unit of partition in this study. However, if some classes are too large for one node to handle and we decide to split them, the heuristics presented in this chapter can still be used to group the partial classes.

It is not easy to find a partition which is good for all the applications. A good partition for one application may not be suitable for another application. If we make a compromise for both applications, neither one will perform well. Therefore,

we decide to partition the database based on the processing requirement of a single application which is characterized by a set of typical queries used in the application. By analyzing the query patterns in the set and the data characteristics of the database, we try to find a partition so that the execution time of the set of queries is minimized.

If we want to calculate the execution time of a query, an appropriate cost function is needed for modeling the parallel execution of the query. For a set of queries, the interaction and interference among queries will make it extremely difficult to formulate the cost function. Even if we can formulate the correct function, the problem of finding the minimum would be intractable. Therefore, instead of finding the best partition which gives the minimal execution time, we try to find some heuristic rules that will avoid bad partitions and give good performance.

6.2 Heuristics for Partitioning an OODB

The execution time of a query in a parallel environment consists of three components: CPU time, IO time, and communication time. Since the CPU time and IO time in each processor are the time the processor works on the query, we use the term "processing cost" to represent these two time components. It takes some communication time for a message to transfer from one processor to another. However, both the source and the destination processors can do other tasks during this time.

If the communication delay is short, one obvious bad solution for partitioning the database is to assign all object classes to one single node. In other words, we want to balance the processing load on the nodes as well as to reduce the communication cost among them. However, we cannot use the sum of the processing cost and the communication cost as the total cost for a partition because it is difficult to give a meaning to the combined cost. Also, if we use the combined cost to partition

the database, we run the risk of having two equal cost partitions in which one has high processing cost and the other one has high communication cost. On the other hand, if the communication delay is long, we want to group classes that exchange large amount of data and reduce the length of the "path" through which messages and data must be transferred. Therefore, we try to find a combined heuristics for partitioning the database.

The heuristic method is based on the overall processing cost of each class referenced in a query. We measure the overall CPU time and IO time used for processing a class to represent the processing cost of that class in the query. When we consider the set of queries, we take the sum of the processing costs for the same class in all queries to represent the total processing cost for that class.

Figure 6.1 shows the example university database with a class number and a class size in parentheses (i.e., the number of instances) attached to each class. A set of 10 queries as shown in Figure 6.2 represents the processing requirement of a specific application that we want to partition the database for. In this example, we have 5 simple queries (queries 0, 1, 2, 3, and 4) and 5 complex queries (queries 5, 6, 7, 8, and 9). Each simple query contains 3 or 4 classes and each complex query has 6 or 7 classes. Since this set of queries has a large variety of query patterns, we feel that it can represent a general application of this database. The number in parentheses beside each class number is the measured processing cost of the class when the query is actually executed on the nCUBE 2 computer. We can calculate the processing cost of each class. For example, class 2 (Transcript) has been referenced twice in query 0 and query 8. The overall processing cost of class 2 in the set is the sum of the processing costs of class 2 in query 0 and query 8. Therefore, the overall processing cost of class 2 is 46.13. The calculated processing costs for all the classes referenced by the query set are shown in Table 6.1.

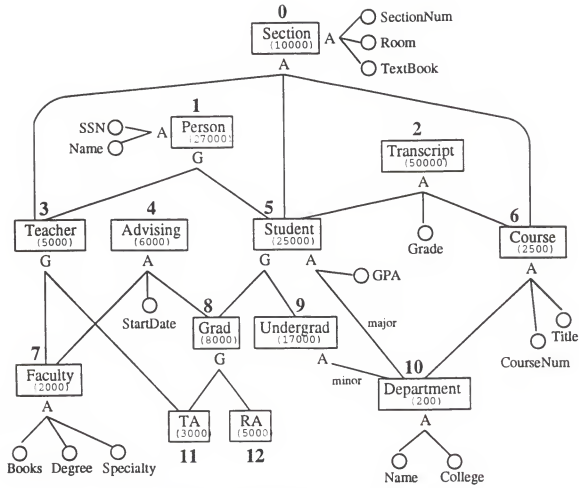
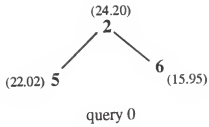
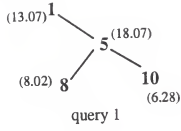


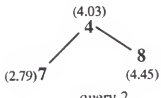
Figure 6.1 The university database.



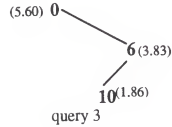
query 0



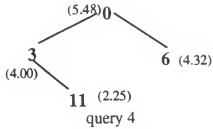
query 1



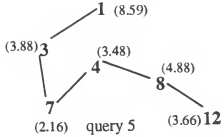
query 2



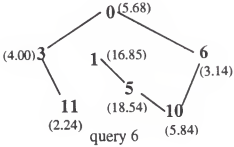
query 3



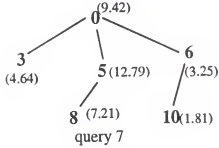
query 4



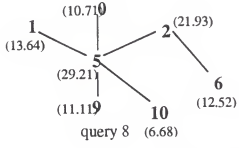
query 5



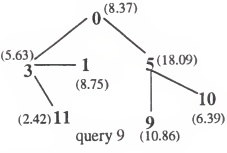
query 6



query 7



query 8



query 9

Figure 6.2 Sample queries

Table 6.1 Processing cost of each class.

Class number	Cost
0	45.26
1	60.90
2	46.13
3	22.15
4	7.51
5	118.72
6	43.01
7	4.95
8	24.56
9	21.97
10	28.86
11	6.91
12	3.66

The overall processing cost of a class represents the minimal work that needs to be done for the set of queries if the class is assigned to a single processor. If we assign multiple classes to a processor, the load of the processor is the sum of the processing costs of the classes that are assigned to it. In order to achieve good performance, we want to distribute the load among the processors as evenly as possible. Load balancing is our main consideration for partitioning the database.

However, when we group two classes and put them on the same processor, the time for exchanging messages between these two classes can be drastically reduced. Therefore, we also want to group classes in a way so that the overall communication time can be reduced. In our multi-wavefront algorithms, wavefronts will be propagated through the longest path in the query pattern. The length of the longest path in a query is called the query diameter. If we reduce the diameter of a query, the overall communication time of the query will also be reduced. The query diameter can be reduced by grouping adjacent classes in the longest path and assigning

them to the same processor. This is our secondary consideration for partitioning the database.

We combine the above two heuristics into the following method for partitioning a database. Since we want to evenly distribute the processing cost among the processors, the number of groups of classes that we formed should be equal to the number of processors, assuming the number of classes is larger than the number of processors. The overall processing cost of each group should be close to the average processing cost among the processors. The average processing cost is called threshold cost. If some of the classes have processing cost that are larger than the average processing cost of the processors, we assign each of them to an empty group and will not assign any other class to these groups. The remaining classes should be distributed among the remaining processors as evenly as possible. Since the processing costs of the classes assigned to the single-class groups are above the threshold cost, the average processing costs of the remaining classes would be lower than the threshold cost. For this reason, we calculate a new threshold cost based on the costs of the remaining classes.

This new threshold cost is used as an upper limit for grouping classes in the first phase. When we group classes together, the total processing cost of the resulting group should not be larger than the new threshold cost. We start from the query with the largest diameter in the set and try to reduce the diameter by grouping two adjacent classes in the longest path. The two adjacent classes with the smallest combined processing cost will be considered. If the combined cost does not exceed the threshold cost, we group them together and use the combined cost as the processing cost of these two classes in all the queries. This step reduces the length of the longest path by 1. Then, we try to reduce the next longest path in the set by 1. If there are multiple paths with the same length, we find a candidate pair of class for

each path and choose the pair with the lowest combined cost to group. This process will continue until we cannot reduce the length of any path by grouping classes or the number of groups is equal to the number of the processors. In this phase, while we group classes to reduce the query diameters, the threshold cost is used to control the load in each group so that we can balance the load during the second phase of our heuristic method.

After we finish grouping classes for reducing query diameters, we need to reduce the number of groups to the number of processors in the system. In other words, we want to form the same number of clusters of groups as the number of processors. First, the groups are sorted based on their processing costs. Then we assign the group with the largest cost to the first available cluster with the lowest cost and add the group's cost to the cluster's cost. By continuing this simple process, we can assign all groups to a fixed number of clusters having relatively close final costs among the clusters. Then, we can assign each cluster to a processor because we assume all the processors are the same and they are fully connected.

6.3 An Example

If we want to partition the university database for a 7-node system, we need to find a suitable set of queries to represent the application and measure the processing cost of each class in each query. An example is show in Figure 6.2. Then, we calculate the overall processing cost of each class and the results are shown in Table 6.1. The next step is to find the threshold cost. Since the total processing cost of all classes is 434.59 and the average processing cost of the 7 processors is 62.08, the cost of class 5 is too large for it to be considered in the following procedure. Therefore, we just assign class 5 to a processor and drop it from further consideration. We also

re-calculate the average cost of the remaining 6 processors and it is 52.65. This is the new threshold cost.

Among the 10 queries, query 6 has the longest diameter of 6. The adjacent classes 3 and 11 have the smallest combined cost (29.06) in the longest path in the query. We group them together. Now, queries 6 and 5 both have a diameter of 5. We check the longest path in query 6 and cannot find two adjacent classes that have a combined cost lower than the threshold cost. In query 5, we find classes 4 and 7 can be grouped together. By continuing this process, we find that the groups with their cost in parentheses are as follows: 5 (118.72); 1 (60.90); 9 and 10 (50.83); 2 (46.13); 0 (45.26); 6 (43.01); 4, 7, 8, and 12 (40.68); 3 and 11 (29.06).

We assign the 7 largest groups to the 7 empty clusters. Then, we assign the next group (in this case 3 and 11) to the lowest cluster. After we finish all the assignment, we have the following clusters: class 5; classes 3, 11, 4, 7, 8, and 12; class 1; classes 9 and 10; class 2; class 0; class 6. The heuristics presented here along with some other methods will be evaluated in the following chapter.

CHAPTER 7
IMPLEMENTATION AND EVALUATION

In this chapter, we present the design and implementation details of a parallel query processor for OODBs on an nCUBE 2 parallel computer. Also, we use this query processor to evaluate multi-wavefront algorithms based on the identification and elimination approaches and to verify the heuristic method for partitioning an OODB.

7.1 Hardware and Software Environment

We have implemented both identification- and elimination-based parallel algorithms for tree-structured queries on a 64-node nCUBE 2 parallel computer. The 64 nodes are connected as a hypercube through communication channels. Each processing node is a sequential computer with its own integrated floating-point unit, local memory, and communication hardware. The CPU is operating at 2.5 MFLOPS or 7.5 MIPS. Every node has 16 MBytes main memory. The DMA communication channels support transfers of data and control messages between any two nodes at 2.2 MBytes/second. Besides the inter-node communication channels, each processing node has a separate channel for IO processors. The IO processors are the same as the processing nodes except that they have additional hardware and software to communicate with disks, displays, and other peripheral devices. A Silicon Graphics workstation served as a host computer is also connected to the nCUBE 2 parallel computer through two IO processors.

A copy of a microkernel, nCX operating system, runs on every nCUBE 2 processor. When a program executes on the nCUBE 2 computer, it executes on a set of

processors linked to form an n-dimensional hypercube, which is called the program's subcube. We use a host-resident utility, ncc, to cross-compile a C program to run on the hypercube array. Then a host-resident utility, xnc, is used to launch the program on the nCUBE 2 computer. The xnc utility allocates a subcube of nodes in the hypercube array, loads the nCUBE executable code onto some or all of these nodes, and translates subsequent IO requests into requests for the host's IO system. Since C++ compiler is unsupported in our system, we choose C as the programming language for implementing the query processor on the nCUBE 2 parallel computer.

The nCUBE 2 parallel computer can run programs that are written in different models, such as UNIX Compatibility Model, Single Program Multiple Data Model, Hosted Model, Heterogeneous Model, Asynchronous Model, and a mixture of the above. Since our multiple wavefront algorithms run on multiple nodes with no constraints on their relative timing, the query processor is structured in an asynchronous model on the nCUBE 2 computer. We use several interprocessor communication routines, such as nwrite, nread, and ntest, provided by the nCUBE Extended Run-Time Library to transmit messages between processors.

7.2 Design and Implementation

Conceptually, a client-server model is used to implement the query processor. As shown in Figure 7.1, a server consists of a master node and several slave nodes which are connected through an interconnection network. In our implementation, we place the master on logical processor 0 in a subcube. The rest of the processors in the subcube are all slaves.

The block diagram of the system is illustrated in Figure 7.2. A database is partitioned using a heuristic method described in the previous chapter and a portion of the database is assigned to each slave node. We decide to store all the connection

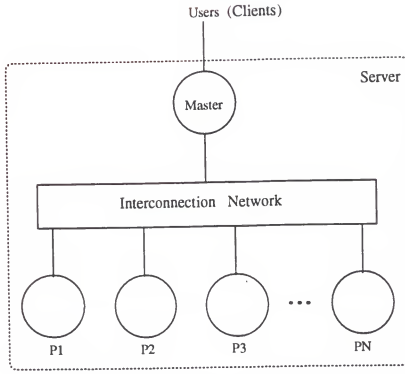


Figure 7.1 Client-server model.

information in the main memory on each node and store the descriptive data on the secondary storage devices. This is because the connection information is frequently accessed during the first phase of our query processing algorithms. Keeping the connection information in the main memory can avoid frequent access to the disks which is very time consuming. Also, the size of the connection information is small enough to be kept in the main memory. For example, a class can contain 5000 instances in a database. On an average, an instance may have 5 connected instances in an associated class. Suppose that it takes 10 bytes to keep the connection information between two connected instances. Therefore, all the connection information about this association needs 250 Kbytes of memory. If each class has 4 associations, the total memory needed for a class is 1 Mbytes. We can keep the connection information of 10 such classes in the main memory on each node in our implementation. In recent years, many researchers have investigated into main memory database

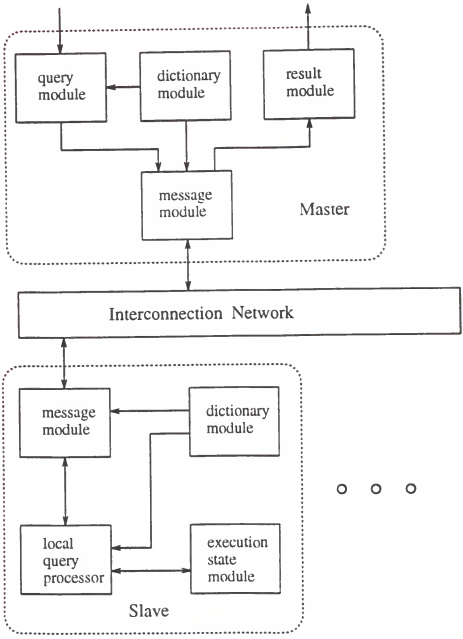


Figure 7.2 System block diagram.

systems. With the rapid advance in memory technology, each processor will have a very large main memory in the near future. We feel that it is reasonable to keep all the connection information in the main memory.

After the database is created, the master receives queries from the user and translates each query into a number of subqueries which are sent to the relevant processors for execution against the parts of the database assigned to them. Concurrently, those processors involved in a query generate the results and send them back to the master. The master in turn presents the results to the user.

Since this system is implemented in an asynchronous model on the nCUBE 2 computer, message passing is the only means that processors can communicate with each other. Therefore, each processor has a message module for handling messages. Database schema and mapping information are stored in a dictionary in every node. The dictionary in a slave node keeps the part of the populated database that is assigned to the node. We shall describe some key software modules in detail in the following subsections.

7.2.1 Message Module

It is possible to have a deadlock situation in a message passing system. For example, if two nodes try to send messages to each other but their input message buffers are full, a deadlock is formed because they cannot send the message. In order to avoid the deadlock situation, we check for input messages before we send out any message. The communication protocol used in the system is shown in Figure 7.3. If there is an input message for the node, we put the message in an input message queue which is a dynamically allocated linked list. Therefore, the number of messages that can be held in the queue is only limited by the size of available local memory. After retrieving all the messages, the node checks an output message queue and sends

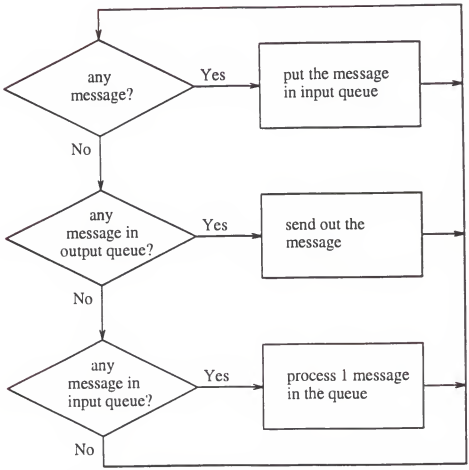


Figure 7.3 Communication protocol block diagram.

out all messages which are waiting in the queue. However, if the node receives new messages, it will put the new messages in the input queue before sending out the next message in the output queue. This guarantees that the incoming messages get treated first, and therefore we can avoid the deadlock situation. After it stores all incoming messages in the input queue and sends out all messages in the output queue, the node will process one received message at a time. This protocol uses a simple first-come, first-served scheduling technique.

Since the message transmitted in the system is a sequence of bytes, we need to transform the message into an internal message structure for further processing. The input and output message queues are linked lists of these message structures. An example of such queue is shown in Figure 7.4. The message structure has 6 data elements: type, from, to, size, cont, and next. The type element is the message type. The from and to elements are the source and destination node numbers. The size element contains the number of bytes in the message content (cont). The next element points to the next message structure in the queue. In this implementation, we have 6 different types of messages. Each type message has its own structure for the message content as illustrated in Figure 7.5. For example, the cont member of a type 1 message points to a sequence of bytes which represents the query number, algorithm number, source class, destination class, number of deleted IIDs, and IIDs. We note that new types of messages can be easily added to this system. Therefore, we can extend this implementation to handle cyclic queries, where conditions, etc.

The message module provides a set of functions for other modules to access data in this module. For example, function insert_message_to_queue() will put a message in the specified queue, and function mywrite() will call the system function nwrite() to send out the message and then free the message pointer. Although we do not use an object-oriented programming language such as C++, we follow the convention

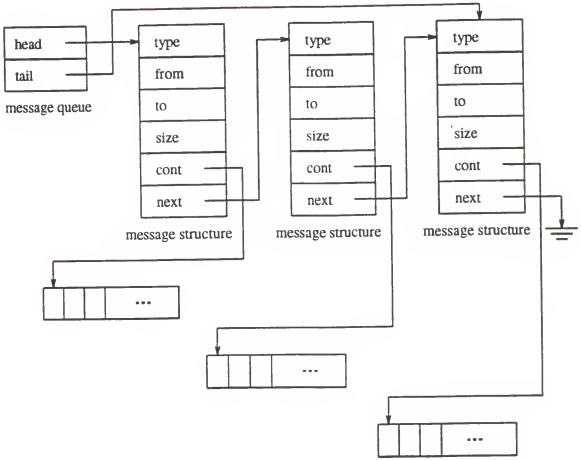


Figure 7.4 A message queue.

type 0 message – query pattern
int: query number,
int: algo,
int: class number,
int: number of end_markers,
int: number of connected classes,
int: connected class, int: connected class, ...

type 1 message – deleted IIDs
int: query number,
int: algo, (should be 1: elimination algorithm)
int: from class number,
int: to class number,
int: number of IIDs,
int: IID, ... (of the from class)

type 2 message – selected IIDs
int: query number,
int: algo, (should be 2: identification algorithm)
int: from class number,
int: to class number,
int: number of IIDs,
int: IID, ... (of the from class)

type 3 message – end_marker
int: query number,
int: algo,
int: from class number,
int: to class number,
int: origin class number of the end_marker,

type 4 message – resulting data (from a node (class) to the master)
int: query number,
int: algo,
int: from class number,
int: number of ints,
int: ints, ... (of the from class)

type 5 message – control messages (from master (node 0) to slave nodes)

Figure 7.5 Message content structures.

of accessing the message module only through these functions. Therefore, we still have the benefit of abstract data types, namely data abstraction and encapsulation, when we develop the programs.

7.2.2 Dictionary Module

The dictionary module contains all the information of the database. As shown in Figure 7.6, the dictionary structure has 2 members which are the number of classes in the schema and a class array pointer. The class array pointer points to a linked list of pairs of a class structure pointer and the assigned node number of the class. Each class structure consists of 5 members: the class number, the number of object instances in the class, a pointer to a list of association structures, a pointer to a list of backward pointer structures, and a pointer to a list of count structures. Each association structure contains the information of one connection in the schema including the association type and a list of connected classes. The backward pointer structure represents the instance level connection information from one connected class, and a count structure contains a CON array to one connected class.

A set of functions are provided in addition to the data structures. For example, init_assoc() will initiate an association structure, insert_assoc_to_class() will insert an association structure into a class structure, and get_class() will return the pointer of the specified class structure. We note that only IIDs and pointers are stored in the dictionary. Therefore it is possible to keep the whole dictionary in the main memory. In the two-phase query processing, only IIDs are processed and propagated in the first phase and the retrieval of sizable descriptive data is postponed until the second phase when the relevant objects have been identified. The descriptive data are stored in the secondary storage devices. Because we are mainly evaluating different query processing strategies for the first phase, we do not implement the database

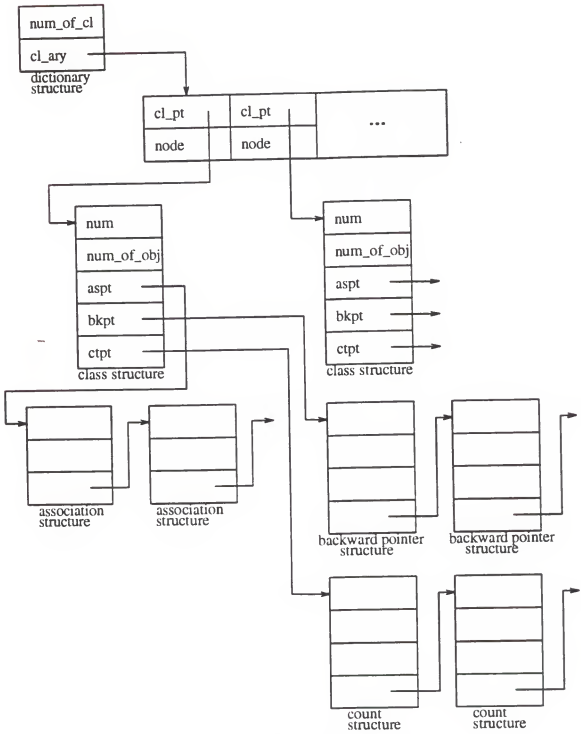


Figure 7.6 Dictionary structure.

on the disks. A distributed object manager which provides the low-level operations has been reported in Bhethanabotla [Bhet92].

7.2.3 Query Module

The query module is only used in the master. After the master gets a set of queries, it stores them in a linked list of query structures. Each query structure contains 4 members: query number, algorithm number, a pointer to a list of operator structures, and a pointer to the next query structure in the list. The operator structure has all the connected classes of one class. The whole list of operator structures represents the query pattern. There is a set of functions for other modules in the master to access data in the query module.

When executing the queries, the master follows a simple FIFO (first-in, first-out) scheduling strategy. It gets the first query in the queue and divides the query into query patterns for each class in the query. Then, the master converts those query patterns into query pattern messages which are ready to be transmitted in the interconnection network. Finally, the query pattern messages are sent to those slave nodes that handle the classes involved in the query according to the class-to-processor mapping information in the dictionary.

Although the queries are sent to slave nodes one by one, they are still executed in parallel in some ways. First, queries may be executed by different sets of nodes and therefore some degree of parallelism can be achieved. Second, several wavefronts may go through a specific class in a query. Between processing wavefronts from the same query, the node can run other queries while it is free. However, if we change the order of queries presented to the master, we may have a different performance because of the new interleaving execution order.

7.2.4 Execution State Module

When a slave node starts to process a query, it creates an execution state for each class to keep track of the execution of the query on that class. The state structure contains information related to the query, the class, status of the execution, and temporary results. Therefore, the node can continue from where the last execution finished after receiving a new message. A set of functions is provided for other modules to access this module.

7.2.5 Result Module

The master uses a result module to store results sent from all slave nodes. This module has a linked list of result structures and each result structure contains the results of one query. A set of functions is also provided for the master to access data in this result module.

7.2.6 Local Query Processor

When a slave node needs to process a received message, it invokes a local query processor. The flow chart for the local query processor is shown in Figure 7.7. Because we implement two algorithms in the query processor, this module checks the algorithm number in the message first. Different sets of functions are used to implement the two algorithms. For the elimination algorithm, three types of message are allowed: a query pattern, deleted IIDs, and an end marker. If the message is a query pattern, we first initiate an execution state for this query on this class. We then check the state and put IIDs that need to be deleted in the output message queue. Also, we put an end marker message to the queue. If the message is a sequence of IIDs, we modify the state according the algorithm presented in the previous chapters. Then, if it is necessary to propagate the wavefront, we insert an IID message into the output message queue. Finally, if the received message

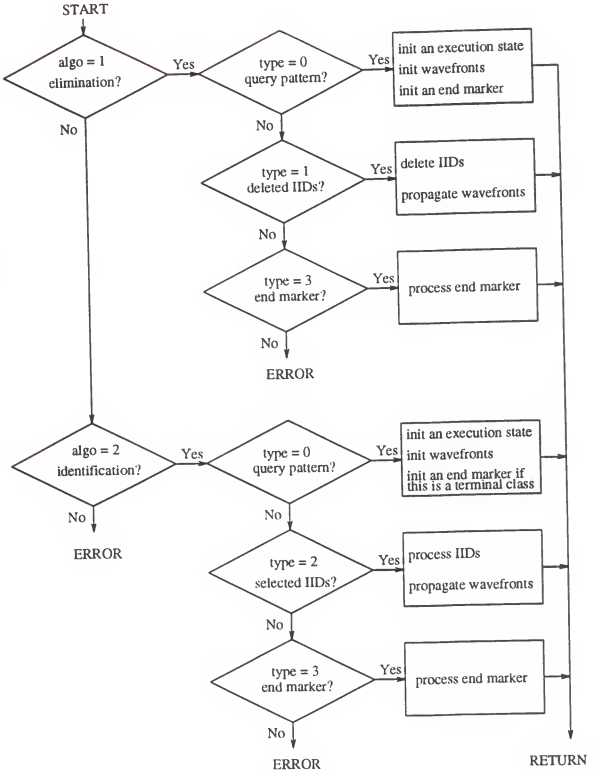


Figure 7.7 Flow chart for the local query processor.

is an end marker message, we add the marker to the state and propagate the end marker to the other connected classes. Also, we check whether the state already has received all the markers. If this is the case, which means that the processing of the query has finished on this class, we will send the results to the master by putting a resulting IID message to the output message queue.

There are also 3 legitimate types of message for the identification algorithm: a query pattern, selected IIDs, and an end marker. For the query pattern message, we first initiate an execution state. If this class is a terminal class in the query pattern, we then initiate identification wavefronts and the end marker. For the IID message, we process the received IIDs and modify the execution state according to the identification algorithm. If it is necessary, we propagate the wavefront to the other connected classes. If the received message is an end marker, we process it as in the elimination algorithm described above.

We note that it is possible for a slave node to receive wavefront messages before it receives the query pattern from the master. Since the query pattern is not available, the local query processor does not know how to process the wavefront messages. Therefore, it just puts the received message back to the output message queue. Then, after the message module puts all input messages into the input message queue, this IID message is also put into the input message queue. Hopefully, by that time, the slave node already receives the query pattern and can process the IID message.

7.3 Generating Test Databases

In order to evaluate the query processor, we need to generate a large database and some sample queries. We use random number generator to create connections between object instances. The university database is used as an example again. As

```
% file: SCHEMA
% date: 5/18/93
% author: Yaw-Huei Chen
%
% This is the university database schema. In this file, each
% line separated by a ";" symbol represents a class. The format
% is as the following:
%
% class number, number of objects (association type, connected
% class number, data file name, connected class number, data
% file name, ...) (...) (...) ;
%
0, 10000 (A, 3, b0, 5, b1, 6, b2);          % Section
1, 27000 (G, 3, b3, 5, b4);                 % Person
2, 50000 (A, 5, b5, 6, b6);                 % Transcript
3, 5000 (G, 7, b7, 11, b8);                 % Teacher
4, 6000 (A, 7, b9, 8, b10);                 % Advising
5, 25000 (G, 8, b11, 9, b12) (A, 10, u13); % Student
6, 2500 (A, 10, b14);                       % Course
7, 2000 ;                                   % Faculty
8, 8000 (G, 11, b15, 12, b16);              % Grad
9, 17000 (A, 10, b17);                      % Undergrad
10, 200;                                    % Department
11, 3000;                                   % TA
12, 5000;                                   % RA
```

Figure 7.8 SCHEMA file.

shown in Figure 7.8, the schema of the university database is specified in the file SCHEMA. The name of a data file is kept along with the connection between two classes. The data file contains the object instance connection information between these two classes. We use another program, generate, to create the data file. The program first collects information from the user: the type of connection, the name of the data file, number of object instances in both classes. constraints on the connection, etc. Then the program randomly generate the data file. For example, the first

non-comment line in the file "0, 10000 (A, 3, b0, 5, b1, 6, b2)" represents the connection information of class Section. The first number, 0, is a unique class number for this class. We use this number to refer to the class throughout the program. The next number, 10000, represents that class Section has 10000 instances. Then, each parenthesis represents a group of associations. The first character in the parenthesis stands for the association type (A for Aggregation and G for Generalization), and the rest are pairs of connected class number and data file name. In this example, class 0 is connected to classes 3, 5, and 6 and the connection information between them are stored in files b0, b1, and b2, respectively.

Comparing to a populated database, the schema information of a database is very small. Therefore, we decide to keep the whole schema information in every node's dictionary. When we launch the query processor on the nCUBE 2 parallel computer, each node will read the SCHEMA file and store the information in its own dictionary structure. However, we have to partition the populated database in order to achieve parallelism. In our implementation, we use a file called ASSIGN to keep the information of class assignment. For example, Figure 7.9 shows such an assignment. In this particular assignment, we have a 3-dimensional subcube allocated for the database. Node 0 is the master. Classes 0 and 1 are assigned to node 1, classes 2 and 3 are assigned to node 2, etc. After the SCHEMA file is read, each node will read the ASSIGN file and populate the classes that are assigned to it. For example, node 1 will only populate classes 0 and 1. This process includes reading the relevant data files and storing the information in the dictionary.

Queries are specified in a query file. The program will read a file called QUERY as the query file, but the user may type in a new file name during the execution. An example of the query file is shown in Figure 7.10. This file contains 5 queries which will be read by the master. Queries are separated by the ";" symbols in the file.

```
% file: ASSIGN
% date: 5/18/93
% author: Yaw-Huei Chen
%
% This is for the university database.
% The first line is the number of classes in this database.
%
13;
%
% Each line represents an assignment. The first number is a class
% number and the second number is the processor which the class
% is assigned to. Do not assign any class to processor 0 because
% processor 0 is reserved for the master.
%
0, 1;
1, 1;
2, 2;
3, 2;
4, 3;
5, 3;
6, 4;
7, 4;
8, 5;
9, 5;
10, 6;
11, 7;
12, 7;
```

Figure 7.9 ASSIGN file.

For each query, the first optional number represents the algorithm that will be used to process this query. If the number is missing, a default algorithm will be used. Then, a set of parenthesis pairs is used to represent the query pattern. There are at least 2 class numbers in each pair of parentheses. They represents connections between the first class to the rest of classes in the parenthesis pair. For example, query 2 in the QUERY file is shown as (3, 0) (0, 5, 6) (5, 8) (6, 10). This means class 3 is connected to class 0, class 0 is connected to classes 5 and 6, etc. After reading in the query file, the master will create a query structure for each query and send them to slave nodes to start the processing. The master will ask the user to input a new query file or to change some system parameters, such as the selectivity factor, when it finishes processing a set of queries. The user may continue or quit the program at this moment.

7.4 Performance Evaluation

We evaluate the performance of the query processor in this section. First, two multi-wavefront algorithms, i.e., identification and elimination, are compared using a 7-class database, which is generated as described in the previous section. In order to reduce the effects from different partitioning strategies. we assign the 7 classes to 7 nodes in a 3-dimensional subcube so that each node only contains a single class. Then, we study the performance of different assignments of physical nodes under this one-class-per-node policy. Finally, the heuristic method for partitioning the database presented in the previous chapter is evaluated. We use the heuristic method to partition the example university database into 7 clusters and assign them to 7 nodes in a 3-dimensional subcube. The performance of these partitions are then evaluated by using a set of benchmark queries.

7.4.1 Comparing the Two Algorithms

Several experiments have been carried out to compare the performance of the identification- and elimination-based algorithms. Some parameters and their default values used in these experiments are as follows. We use 7 classes each of which has 2,000 object instances as the part of the schema referenced by queries. We feel that seven classes are adequate to test a wide range of queries since a 7-class query is a rather complex query. In our experiment, we vary the query diameter (maximal path length in the query graph) to test the effects of different query patterns. The default query diameter is assumed to be 6, i.e., the 7 classes are linearly connected. The default data connectivity is 2, which means that, on an average, an object instance of a class is related to 2 instances of an associated class. Since the data connectivity is varied from 0.5 to 5 in the experiment, we in fact have tested a wide range of database sizes even though the number of instances per class is fixed at 2,000. The measurement value that we present here is the mean value of 3 measurements. The standard deviation of the measurements is very small (less than 5%) and can be neglected.

The execution time that we measured in these experiments consists of 3 components: CPU processing time, communication time, and disk IO time. We start to measure the time when the master sends query patterns to all the slave nodes that are involved in the query and measure the finish time after the master receives results from all those slave nodes. After a slave node receives the query pattern from the master, it checks the local selection condition by reading the relevant descriptive data from the disk. Since the object manager is not implemented on the nCUBE 2 computer, we simulate this operation by reading dummy data from a disk. It reads 10 bytes from the disk for each instance in the class. On the nCUBE 2 computer, each node can read any one of the disks in the system. In order to reduce

```
% file: QUERY
% date: 5/18/93
% author: Yaw-Huei Chen
%
% This file contains a batch of queries. Queries are separated by
% ";" symbols. The first digit in each query represents the algorithm
% used to process the query. If the first digit is omitted, a default
% algorithm will be used. Each query has several ( ) pairs. For
% each ( ) pair, say (2, 3, 6), the first number 2 is the starting
% class number and the rest of the numbers, 3 and 6, are classes that
% are connected to the first class, 2.
%
% query 0: 1 * 3 * 7 * 4 * 8 * 12
(1,3) (3,7) (7,4) (4,8) (8,12);
%
% query 1: 11 * 3 * 0 * 6 * 10 * 5 * 1
(11,3) (3,0) (0,6) (6,10) (10,5) (5,1);
%
% query 2: 3 * 0 * AND((5 * 8), (6 * 10))
(3,0) (0, 5. 6) (5, 8) (6, 10);
%
% query 3: 1 * 5 * AND(0, (2 * 6), 9, 10)
(1,5) (5,0,2,9,10) (2,6);
%
% query 4: 11 * 3 * AND(1, (0 * 5 * AND(9, 10)))
(11,3) (3,1,0) (0,5) (5,9,10);
```

Figure 7.10 QUERY file.

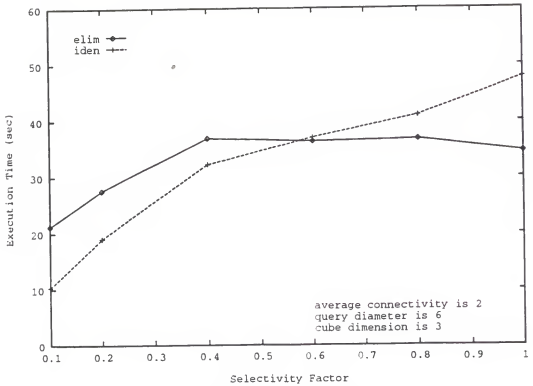


Figure 7.11 Execution time vs. selectivity factor.

the interference from other nodes, we assign a dedicated disk for each node in our implementation. Then, the slave nodes start the wavefront algorithm by sending and processing messages. When the algorithm is finished on a class, the node that handles the class will read the descriptive data of those selected instances in the class. Again, this operation is simulated by reading dummy data from a disk to get the actual disk IO time. Since we assume that the user needs all the data of the instance, the node reads 100 bytes from the disk for each selected instance. Then, the slave node sends the results to the master.

The selectivity factor is the ratio of instances in each class that satisfy the local selection condition before the algorithm starts matching the query pattern. Figure 7.11 shows the total execution time for both algorithms as the selectivity factor is varied from 0.1 to 1.0. For the identification algorithm, a higher selectivity

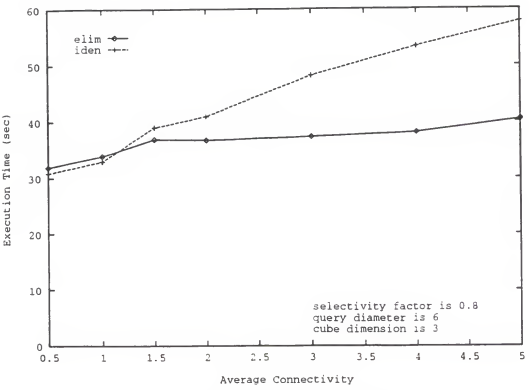


Figure 7.12 Execution time vs. connectivity.

factor means that more instances satisfy local selection. This implies more processing time, communication time, and disk IO time. For the elimination algorithm, a higher selectivity factor implies that less instances need to be deleted so that the processing time is lower. However, the disk IO time will increase as the selectivity factor increases. The default selectivity factor is 0.8 in the following experiments.

Figure 7.12 shows the execution time as the average connectivity of each instance is varied. The processing time of both algorithms increase with increasing connectivities. When the average connectivity is low, a small number of object instances in every class may satisfy the query pattern. On the other hand, there is a large number of object instances in each class that satisfy the query pattern if the average connectivity is high. For the identification algorithm, a large number of selected instances means more processing, communication, and disk IO time and

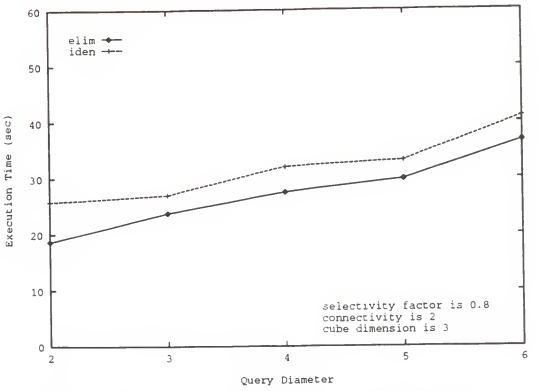


Figure 7.13 Execution time vs. query diameter of linear queries.

vice versa. However, since the elimination algorithm deletes instances that do not satisfy the query pattern, a large number of selected instances implies less number of IIDs that need to be processed and sent out. This compensates the effects of large average connectivity and therefore the total execution time of the elimination algorithm is lower than the identification algorithm. The opposite situation also explains the higher execution time of the elimination algorithm when the average connectivity is low.

Different query patterns may affect the algorithms' performance. We show in Figure 7.13 the execution time of linear queries as the query diameter is varied. Since all queries tested in this experiment have linear patterns, they have different number of classes specified in the queries. For example, a 3-class linear query has a query diameter of 2 and a 4-class linear query has a query diameter of 3, etc.

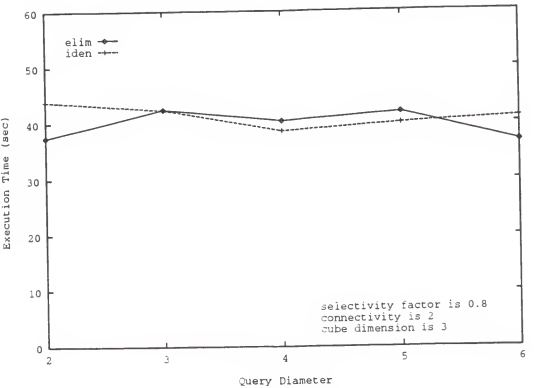


Figure 7.14 Execution time vs. query diameter of 7-class queries.

When the query diameter is short, both algorithms perform well. This is because the query with short diameter has less data to process and a shorter propagation delay.

Figure 7.14 shows the execution time of 7-class queries with different query diameters. Generally speaking, when the diameter is long, there will be a longer IID propagation delay. When the diameter is short and the number of classes is kept constant, the query will have more branches and the nodes with branches will have more data to work on. So the performance of a query varies depending on the trade-off between the propagation delay and the possible overloading of those processors with branches. However, it is clear that the identification algorithm suffers from a bottleneck situation in a 7-class query with a query diameter of 2.

From the above experiment results, it can be observed that the parallel elimination algorithm is better in more cases, especially when the query diameter is short. However, the identification algorithm is better if a small number of objects participate in the identification process. It is important to note that neither approach is always better; they should be considered as complementary.

7.4.2 Testing Different Assignments

In the following experiment we study the effects of different assignments under the one-class-per-node policy. It is obvious that the communication between physically linked nodes takes less time than that of distant nodes. Therefore, we try to find an assignment which keeps directly connected classes in directly linked nodes. However, this requirement is not important for this particular implementation. This is because the communication channels on the nCUBE 2 computer "support transfers of data and control messages between any two nodes at 2.2 MBytes/second. Even though direct hardware channels exist only between nearest neighbors in the network, the speed of the routing hardware makes communication between distant nodes almost as fast as that between nearest neighbors" [nCU92].

Figure 7.15 shows the execution time of the identification algorithm under different assignments on a 6-dimensional subcube. The average hop is the average distance of classes in the assignment. If the average hop is 1, all the directly connected classes are kept in neighboring nodes. Intuitively, this one hop assignment should have the smallest execution time. However, this is not true when the algorithm needs to send a large amount of data to the master node (i.e., with IO). The classes are close to each other under the small hop number assignments, and this type of assignment may cause communication bottlenecks when all slave nodes try to send data to the master node. On the other hand, When the algorithm only

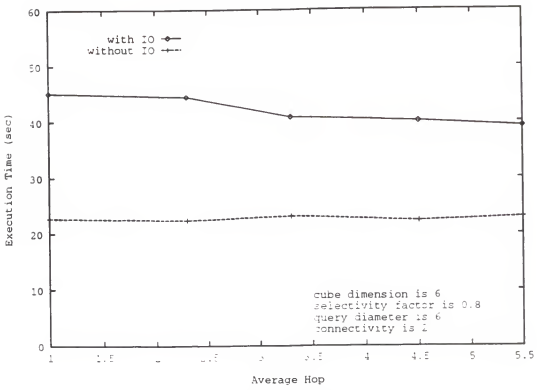


Figure 7.15 Execution time of different assignments.

verifies the query pattern in each class but does not send descriptive data to the master node (i.e., without IO), the execution time of different assignments is almost the same. The elimination algorithm also has a similar performance under different assignments. These results show that different assignments do not affect the performance very much in this implementation on the nCUBE 2 computer and we may assign a class to any node available in the network.

7.4.3 Evaluating Partition Heuristics

Some heuristics for partitioning a database for a parallel computer environment have been proposed in the previous chapter. We use the query processor implemented on the nCUBE 2 parallel computer to evaluate those heuristics. The test database is shown in Figure 6.1. The 10 queries which represent the application is shown in Figure 6.2. Each query is executed independently to collect the processing cost as shown in the figure. Using the processing cost information, a partition is generated by a heuristic method as described in the previous chapter.

We try to map the database onto a 7-node system. Because the assignment of physical nodes does not make much difference as we described in the previous section, we want to find the best way of partitioning the database into groups of object classes and then randomly assign them to the nodes. In order for us to test the effects of different communication delays, we introduce a communication delay variable to the message module in the implementation. Therefore, we can specify the amount of communication delay and the system will wait approximate that amount of time before it sends out the message. However, the processor will not be tied up by the waiting loop, and it can still read messages and process them. The total execution time in these experiments includes 3 time components: CPU time, communication time, and disk IO time. Figure 7.16 shows the experiment results

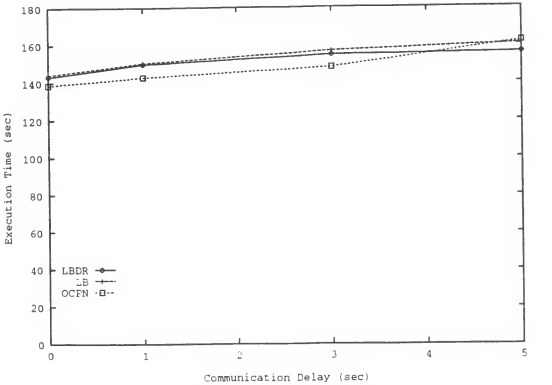


Figure 7.16 Results of heuristic partitions.

of 3 different partitions. The LBDR heuristic method is described in the previous chapter. This method first uses a threshold cost as a guide to group small classes so that the query diameters can be reduced. Then, it tries to evenly distribute the cost among all the processors. The LB heuristic method does not try to reduce the query diameters in the query set. It directly goes to the second step and tries to balance the load. The one-class-per-node partitioning method (OCPN) assigns only one class to a node. Since we have 13 classes, this partition uses a 4-dimensional subcube. It is clearly shown in the figure that the partition of the LBDR method performs better than the LB method when the communication delay increases. Even when the communication delay is small the LBDR performs well comparing to the LB method. This means the heuristics used for partition the database is a good one.

Table 7.1 Experiment results of different partitions.

	OCPN	LBDR	LB	GENE	RANDOM
Execution time (sec)	138.52	143.10	144.17	155.46	185.98

Besides the two heuristic and one-class-per-node partitioning methods, we also test some other partitioning methods. The generalization partitioning method (GENE) in favor of grouping classes that are connected through the generalization association in the schema graph. This is because the super-class and sub-class are frequently accessed together due to inference. By grouping them together, we try to reduce the communication time between them. The random partitioning method (RANDOM) assigns classes to nodes at random. We use the set of 10 queries to test these partition strategies. Table 7.1 shows the experiment results of these different partitions. The results show that the performance of the heuristic method (LBDR) is close to the the one-class-per-node partition (OCPN) on a 4-dimensional subcube, which is the best performance if the communication delay is small.

CHAPTER 8
CONCLUSION

Object-oriented database management systems and their underlying models possess some desirable features that are required for modeling and processing complex objects found in today's data-intensive applications. However, due to the generality and high functionality of these systems, the performance of large object-oriented databases is often limited by the sequential nature of conventional computer systems. Thus, parallel algorithms and architectures are needed to support OODB processing.

In this dissertation, we have presented parallel query processing techniques for OODBs. Two pattern-based and two-phase query processing approaches, namely identification and elimination approaches, have been introduced for verifying object association patterns. A formal graph model is used to transform the query processing problem into a graph problem. Based on the graph model, we proved the correctness of both approaches for tree-structured queries. Also, a combined approach for solving cyclic queries is provided and proved. Identification- and elimination-based parallel and multi-wavefront algorithms have been introduced for both acyclic and cyclic queries. These algorithms can have more processors operating concurrently on a query than the traditional tree-based query processing approach. Thus, a higher degree of parallelism in query processing can be achieved.

We have proposed a heuristic method for partitioning the database. The database is partitioned for a specific application the processing requirement of which is

represented by a set of queries. By analyzing the queries and the system characteristics, we can partition the database to suit the application. We have implemented a query processor using the identification and elimination approaches on an nCUBE 2 parallel computer. The implementation of the query processor allows multiple queries to be executed simultaneously. This implementation provides an environment for evaluating the algorithms and the heuristic method for partitioning the database.

Since the elimination approach can start the wavefronts from all the classes in a query at the same time, it allows a higher degree of parallelism in query processing. The experiment results show that the elimination algorithm is better than the identification algorithm in more cases. However, neither approach is always better. We need to consider them as complementary. The heuristic method that we proposed for partitioning an OODB tries to distribute the processing costs among the processors as evenly as possible. It also tries to reduce query diameters so that the communication costs can be reduced. The experiment results show that the partition generated by this heuristic method provides good performance comparing to other partitioning strategies.

Some future research directions for parallel query processing are outlined as follows:

1. In this dissertation, we have discussed the possibility of starting the wavefronts from a small number of classes. We have shown that this technique can drastically reduce the amount of data that need to be processed in some cases. Query optimization based on this technique needs to be further studied.

2. In the traditional tree-based query processing, a query tree is evaluated in a sequential order from leaves to the root. Query optimization techniques are

developed for manipulating the query tree. Since a graph-based asynchronous query processing strategy is used in this research, new optimization techniques need to be developed to take advantage of this strategy.

3. Since neither the identification approach nor the elimination approach always performs better than the other, the query processor needs to be modified so that it can select either approach to suit some specific database and query characteristics.

4. When multiple queries are executed in the system concurrently, the order of execution can make a difference in performance. Scheduling the query execution is another topic that needs to be further studied.

5. A system can gather the query statistics such as query types and frequencies in an application domain and use the statistics to partition the database following the heuristics described in this work. Re-partitioning of a database can be carried out periodically as the query statistics change from time to time as the processing requirement of an application changes.

REFERENCES

[Aho74] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. The Design and Analysis of Computer Algorithms. Addison-Wesley, Reading, MA, 1974.

[Alas89] A. M. Alashqur, S. Y. W. Su, and H. Lam. OQL: A query language for manipulating object-oriented databases. In Proc. 15th Int'l Conf. on Very Large Data Bases, Amsterdam, The Netherlands, pp. 433–442, Aug. 1989.

[Alas90] A. M. Alashqur, S. Y. W. Su, and H. Lam. A rule-based language for deductive object-oriented databases. In Proc. 6th Int'l Conf. on Data Eng., Los Ansgeles, CA, pp. 58–67, Feb. 1990.

[Alex88] W. Alexander and G. Copeland. Process and dataflow control in distributed data-intensive systems. In Proc. ACM SIGMOD Int'l Conf. on Management of Data, Chicago, IL, pp. 90–98, June 1988.

[Aper83] P. M. G. Apers, A. R. Hevner, and S. B. Yao. Optimization algorithms for distributed queries. IEEE Trans. Softw. Eng., SE-9(1):57–68, Jan. 1983.

[Bern81] P. A. Bernstein and D. W. Chiu. Using semi-joins to solve relational queries. J. ACM, 28(1):25–40, Jan. 1981.

[Bhet92] S. S. Bhethanabotla. Design and implementation of a distributed object manager. Master's thesis, Department of Electrical Engineering, University of Florida, Gainesville, 1992.

[Bic89] L. Bic and R. L. Hartmann. AGM: A dataflow database machine. ACM Trans. Database Syst., 14(1):114–146, Mar. 1989.

[Care88] M. J. Carey, D. J. DeWitt, and S. L. Vandenberg. A data model and query language for EXODUS. In Proc. ACM SIGMOD Int'l Conf. on Management of Data, Chicago, IL, pp. 413–423, June 1988.

[Care90] M. Carey, E. Shekita, G. Lapis, B. Lindsay, and J. McPherson. An incremental join attachment for Starburst. In Proc. 16th Int'l Conf. on Very Large Data Bases, Brisbane, Australia, pp. 662–673, Aug. 1990.

[Chak86] U. S. Chakravarthy and J. Minker. Multiple query processing in deductive databases using query graphs. In Proc. 12th Int'l Conf. on Very Large Data Bases, Kyoto, Japan, pp. 384–391, Aug. 1986.

[Chak91] S. Chakravarthy. Divide and conquer: A basis for augmenting a conventional query optimizer with multiple query processing capabilities. In Proc. 7th Int'l Conf. on Data Eng., Kobe, Japan, pp. 482–490, Apr. 1991.

[Chan77] A. K. Chandra and P. M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In Conf. Record of the 9th Annual ACM Symp. on Theory of Computing, Boulder, CO, pp. 77–90, May 1977.

[Chen89] A. L. P. Chen, D. Brill, M. Templeton, and C. T. Yu. Distributed query processing in a multiple database system. IEEE Journal on Selected Areas in Communications, 7(3):390–398, Apr. 1989.

[Chen91a] M.-S. Chen and P. S. Yu. Determining beneficial semijoins for a join sequence in distributed query processing. In Proc. 7th Int'l Conf. on Data Eng., Kobe, Japan, pp. 50–58, Apr. 1991.

[Chen91b] J. R. Cheng and A. R. Hurson. Effective clustering of complex objects in object-oriented databases. In Proc. ACM SIGMOD Int'l Conf. on Management of Data, Denver, CO, pp. 22–31, May 1991.

[Cope85] G. Copeland and S. Khoshafian. A decomposition storage model. In Proc. ACM SIGMOD Int'l Conf. on Management of Data, Austin, TX, pp. 268–279, May 1985.

[Cope88] G. Copeland, W. Alexander, E. Boughter, and T. Keller. Data placement in Bubba. In Proc. ACM SIGMOD Int'l Conf. on Management of Data, Chicago, IL, pp. 99–108, June 1988.

[Denn80] J. B. Dennis. Data flow supercomputers. IEEE Computer, 13(11):48–56, Nov. 1980.

[Deux90] O. Deux et al. The story of O_2. IEEE Trans. Knowledge Data Eng., 2(1):91–108, Mar. 1990.

[DeWi84] D. J. DeWitt, R. H. Katz, F. Olken, L. D. Shapiro, M. R. Stonebraker, and D. Wood. Implementation techniques for main memeory database systems. In Proc. ACM SIGMOD Int'l Conf. on Management of Data, Boston, MA, pp. 1–8, June 1984.

[DeWi86] D. J. DeWitt, R. Gerber, G. Graefe, M. Heytens, K. Kumar, and M. Muralikrishna. GAMMA–A high performance dataflow database machine. In Proc. 12th Int'l Conf. on Very Large Data Bases, Kyoto, Japan, pp. 228–237, Aug. 1986.

[DeWi90] D. J. DeWitt, S. Ghandeharizadeh, D. A. Schneider, A. Bricker, H.-I. Hsiao, and R. Rasmussen. The Gamma database machine project. IEEE Trans. Knowledge Data Eng., 2(1):44–62, Mar. 1990.

[Dowd82] L. W. Dowdy and D. V. Foster. Comparative models of the file assignment problem. ACM Comput. Surv., 14(2):287–313, June 1982.

[Fink82] S. Finkelstein. Common expression analysis in database applications. In Proc. ACM SIGMOD Int'l Conf. on Management of Data, Orlando, FL, pp. 235–245, June 1982.

[Gaud91] J.-L. Gaudiot and L. Bic, editors. Advanced Topics in Data-Flow Computing. Prentice Hall, Englewood Cliffs, NJ, 1991.

[Ghan90a] S. Ghandeharizadeh and D. J. DeWitt. Hybrid-range partitioning strategy: A new declustering strategy for multiprocessor database machines. In Proc. 16th Int'l Conf. on Very Large Data Bases, Brisbane, Australia, pp. 481–492, Aug. 1990.

[Ghan90b] S. Ghandeharizadeh and D. J. DeWitt. A multiuser performance analysis of alternative declustering strategies. In Proc. 6th Int'l Conf. on Data Eng., Los Angeles, CA, pp. 466–475, Feb. 1990.

[Grae90] G. Graefe. Encapsulation of parallelism in the Volcano query processing system. In Proc. ACM SIGMOD Int'l Conf. on Management of Data, Atlantic City, NJ, pp. 102–111, June 1990.

[Guo91] M. Guo, S. Y. W. Su, and H. Lam. An association algebra for processing object-oriented databases. In Proc. 7th Int'l Conf. on Data Eng., Kobe, Japan, pp. 23–32, Apr. 1991.

[Haas90] L. M. Haas, W. Chang, G. M. Lohman, J. McPherson, P. F. Wilms, G. Lapis, B. Lindsay, H. Pirahesh, M. J. Carey, and E. Shekita. Starburst mid-flight: As the dust clears. IEEE Trans. Knowledge Data Eng., 2(1):143–160, Mar. 1990.

[Hall76] P. A. V. Hall. Optimization of single expressions in a relational data base system. IBM J. Res. Dev., 20(3):244–257, May 1976.

[Hara92] L. Harada and N. Akaboshi. Evaluation results of multi-way joins in shared-nothing database environment–The case for left-deep and right-deep query execution tree. Fujitsu Laboratories. Kawasaki, Japan, 1992.

[Ibar84] T. Ibaraki and T. Kameda. On the optimal nesting order for computing n-relational joins. ACM Trans. Database Syst., 9(3):482–502, Sept. 1984.

[Ioan87] Y. E. Ioannidis and E. Wong. Query optimization by simulated annealing. In Proc. ACM SIGMOD Int'l Conf. on Management of Data, San Francisco, CA, pp. 9–22, May 1987.

[Ioan90] Y. E. Ioannidis and Y. C. Kang. Randomized algorithms for optimizing large join queries. In Proc. ACM SIGMOD Int'l Conf. on Management of Data, Atlantic City, NJ, pp. 312–321, May 1990.

[Ioan91] Y. E. Ioannidis and Y. C. Kang. Left-deep vs. bushy trees: An analysis of strategy spaces and its implications for query optimization. In Proc. ACM SIGMOD Int'l Conf. on Management of Data, Denver, CO, pp. 168–177, May 1991.

[Jark84] M. Jarke and J. Koch. Query optimization in database systems. ACM Comput. Surv., 16(2):111–152, June 1984.

[Jark85] M. Jarke. Common subexpression isolation in multiple query optimization. In W. Kim, D. S. Reiner, and D. S. Batory, editors, Query Processing in Database Systems, pp. 191–205. Springer-Verlag, Berlin, 1985.

[Jenq90] B. P. Jenq, D. Woelk, W. Kim, and W.-L. Lee. Query processing in distributed ORION. In F. Bancilhon, C. Thanos, and D. Tsichritzis, editors, Advances in Database Technology – EDBT '90, pp. 169–187. Springer-Verlag, Berlin, 1990.

[Kamb82] Y. Kambayashi, M. Yoshikawa, and S. Yajima. Query processing for distributed database using generalized semijoins. In Proc. ACM SIGMOD Int'l Conf. on Management of Data, pp. 151–160, June 1982.

[Kamb85] Y. Kambayashi. Processing cyclic queries. In W. Kim, D. S. Reiner, and D. S. Batory, editors, Query Processing in Database Systems, pp. 62–78. Springer-Verlag, Berlin, 1985.

[Kell91] T. Keller, G. Graefe, and D. Maier. Efficient assembly of complex objects. In Proc. ACM SIGMOD Int'l Conf. on Management of Data, Denver, CO, pp. 148–157, May 1991.

[Khos87] S. Khoshafian, G. Copeland, T. Jagodits, H. Boral, and P. Valduriez. A query processing strategy for the decomposed storage model. In Proc. 3rd Int'l Conf. on Data Eng., Los Angeles, CA, pp. 636–643, Feb. 1987.

[Khos88] S. Khoshafian, P. Valduriez, and G. Copeland. Parallel query processing for complex objects. In Proc. 4th Int'l Conf. on Data Eng., Los Angeles, CA, pp. 202–209, Feb. 1988.

[Kim85] W. Kim, D. S. Reiner, and D. S. Batory, editors. Query Processing in Database Systems. Springer-Verlag, Berlin, 1985.

[Kim90] W. Kim, J. F. Garza, N. Ballou, and D. Woelk. Architecture of the ORION next-generation database system. IEEE Trans. Knowledge Data Eng., 2(1):109–124, Mar. 1990.

[Kits84] M. Kitsuregawa, H. Tanaka, and T. Moto-oka. Architecture and performance of relational algebra machine GRACE. In Proc. of the Int'l Conf. on Parallel Processing, Bellaire, MI, pp. 241–250, Aug. 1984.

[Kits90] M. Kitsuregawa and Y. Ogawa. Bucket spreading parallel hash: A new, robust, parallel hash join method for data skew in the super database computer (SDC). In Proc. 16th Int'l Conf. on Very Large Data Bases, Brisbane, Australia, pp. 210–221, Aug. 1990.

[Kris86] R. Krishnamurthy, H. Boral, and C. Zaniolo. Optimization of nonrecursive queries. In Proc. 12th Int'l Conf. on Very Large Data Bases, Kyoto, Japan, pp. 128–137, Aug. 1986.

[Lam87] H. Lam, S. Y. W. Su, F. L. C. Seeger, C. Lee, and W. R. Eisenstadt. A special function unit for database operations within a data-control flow system. In Proc. of the Int'l Conf. on Parallel Processing, Chicago, IL, pp. 330–339, Aug. 1987.

[Lam89] H. Lam, C. Lee, and S. Y. W. Su. An object flow computer for database applications. In Proc. of the Int'l Workshop on Database Machines, Deauville, France, pp. 1–17, June 1989.

[Lu91] H. Lu, M.-C. Shan, and K.-L. Tan. Optimization of multi-way join queries for parallel execution. In Proc. 17th Int'l Conf. on Very Large Data Bases, Barcelona, Spain, pp. 549–560, Sept. 1991.

[Maie83] D. Maier. The Theory of Relational Databases. Computer Science Press, Rockville, MD, 1983.

[nCU92] nCUBE, nCUBE 2 Programmer's Guide. Release 3.0. nCUBE, Foster City, CA, 1992.

[Park88] J. Park and A. Segev. Using common subexpressions to optimize multiple queries. In Proc. 4th Int'l Conf. on Data Eng., Los Angeles, CA, pp. 311–319, Feb. 1988.

[Rose88] A. Rosenthal and U. S. Chakravarthy. Anatomy of a modular multiple query optimizer. In Proc. 14th Int'l Conf. on Very Large Data Bases, Los Angeles, CA, pp. 230–239, Aug.-Sept. 1988.

[Rous91] N. Roussopoulos and H. Kang. A pipeline N-way join algorithm based on the 2-way semijoin program. IEEE Trans. Knowledge Data Eng., 3(4):486–495, Dec. 1991.

[Schn89] D. A. Schneider and D. J. DeWitt. A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment. In Proc. ACM SIGMOD Int'l Conf. on Management of Data, Portland, OR, pp. 110–121, June 1989.

[Schn90] D. A. Schneider and D. J. DeWitt. Tradeoffs in processing complex join queries via hashing in multiprocessor database machines. In Proc. 16th

Int'l Conf. on Very Large Data Bases, Brisbane, Australia, pp. 469–480, Aug. 1990.

[Seli79] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In Proc. ACM SIGMOD Int'l Conf. on Management of Data, Boston, MA, pp. 23–34, May-June 1979.

[Sell88] T. K. Sellis. Multiple-query optimization. ACM Trans. Database Syst., 13(1):23–52, Mar. 1988.

[Shan91] K. Shannon and R. Snodgrass. Semantic clustering. In A. Dearle, G. M. Shaw, and S. B. Zdonik, editors, Implementing Persistent Object Bases: Principles and Practice, pp. 389–402. Morgan Kaufmann Publishers, Palo Alto, CA, 1991.

[Shap86] L. D. Shapiro. Join processing in database systems with large main memories. ACM Trans. Database Syst., 11(3):239–264, Sept. 1986.

[Shek90] E. J. Shekita and M. J. Carey. A performance evaluation of pointer-based joins. In Proc. ACM SIGMOD Int'l Conf. on Management of Data, Atlantic City, CA, pp. 300–311, May 1990.

[Smit75] J. M. Smith and P. Y.-T. Chang. Optimizing the performance of a relational algebra database interface. Commun. ACM, 18(10):568–579, Oct. 1975.

[Su86] S. Y. W. Su, K. P. Mikkilineni, R. A. Liuzzi, and Y. C. Chow. A distributed query processing strategy using decomposition, pipelining and intermediate result sharing techniques. In Proc. 2nd Int'l Conf. on Data Eng., Los Angeles, CA, pp. 94–102, Feb. 1986.

[Su89] S. Y. W. Su, V. Krishnamurthy, and H. Lam. An object-oriented semantic association model (OSAM*). In S. Kumara, A. L. Soyster, and R. L. Kashyap, editors, Artificial Intelligence: Manufacturing Theory and Practice, pp. 463–494. Institute of Industrial Engineers, Industrial Engineering and Management Press, Norcross, GA, 1989.

[Su91] S. Y. W. Su, Y.-H. Chen, and H. Lam. Multiple wavefront algorithms for pattern-based processing of object-oriented databases. In Proc. 1st Int'l Conf. on Parallel and Distributed Inf. Syst., Miami Beach, FL, pp. 46–55, Dec. 1991.

[Swam88] A. Swami and A. Gupta. Optimization of large join queries. In Proc. ACM SIGMOD Int'l Conf. on Management of Data, Chicago, IL, pp. 8–17, June 1988.

[Swam89] A. Swami. Optimization of large join queries: Combining heuristics and combinatorial techniques. In Proc. ACM SIGMOD Int'l Conf. on Management of Data, Los Angeles, CA, pp. 367–376, June 1989.

[Tay89] Y. C. Tay. Attribute agreement. In Proc. of the 8th ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems, Philadelphia, PA, pp. 110–119, Mar. 1989.

[Thak90a] A. K. Thakore. Data Distribution and Algorithms for Asynchronous Parallel Processing of Object-Oriented Knowledge Bases. Ph.D. dissertation, Department of Electrical Engineering, University of Florida, Gainesville, 1990.

[Thak90b] A. K. Thakore, S. Y. W. Su, H. Lam, and D. G. Shea. Asynchronous parallel processing of object bases using multiple wavefronts. In Proc. of the Int'l Conf. on Parallel Processing, St. Charles, IL, pp. 127–135, Aug. 1990.

[Trel82] P. C. Treleaven, D. R. Brownbridge, and R. P. Hopkins. Data-driven and demand-driven computer architecture. ACM Comput. Surv., 14(1):93–143, Mar. 1982.

[Tsan91] M. M. Tsangaris and J. F. Naughton. A stochastic approach for clustering in object bases. In Proc. ACM SIGMOD Int'l Conf. on Management of Data, Denver, CO, pp. 12–21, May 1991.

[Ullm89] J. D. Ullman. Principles of Database and Knowledge-Base Systems, Volume II. The New Technologies. Computer Science Press, Rockville, MD, 1989.

[Vald84] P. Valduriez and G. Gardarin. Join and semijoin algorithms for a multiprocessor database machine. ACM Trans. Database Syst., 9(1):133–161. Mar. 1984.

[Vald87] P. Valduriez. Join indices. ACM Trans. Database Syst., 12(2):218–246, June 1987.

[Wang91] C. Wang, V. O. K. Li, and A. L. P. Chen. Distributed query optimization by one-shot fixed-precision semi-join execution. In Proc. 7th Int'l Conf. on Data Eng., Kobe, Japan, pp. 756–763, Apr. 1991.

[Wilk90] K. Wilkinson, P. Lyngbæk, and W. Hasan. The Iris architecture and implementation. IEEE Trans. Knowledge Data Eng., 2(1):63–75, Mar. 1990.

[Wils91] A. N. Wilschut and P. M. G. Apers. Dataflow query execution in a parallel main-memory environment. In Proc. 1st Int'l Conf. on Parallel and Distributed Inf. Syst., Miami Beach, FL, pp. 68–77, Dec. 1991.

[Wong76] E. Wong and K. Youssefi. Decomposition – a strategy for query processing. ACM Trans. Database Syst., 1(3):223–241, Sept. 1976.

[Yu84] C. Yu and C. Chang. Distributed query processing. ACM Comput. Surv., 16(4):399–433, Dec. 1984.

[Zani83] C. Zaniolo. The database language GEM. In Proc. ACM SIGMOD Int'l Conf. on Management of Data, San Jose, CA, pp. 207–218, May 1983.

BIOGRAPHICAL SKETCH

Yaw-Huei Chen was born in 1959, in Taiwan, Republic of China. He received the B.S. degree in electrical engineering from the National Taiwan University in 1982. After two years of service in the military in Taiwan, he attended the University of Arizona at Tucson, Arizona, and received the M.S. degree in electrical engineering from that institution in 1987. After graduation from UA, Mr. Chen continued his doctoral study in the Department of Computer and Information Sciences at the University of Florida. He has been a research assistant in the Database Systems Research and Development Center at the University of Florida since 1988. He will graduate with the Ph.D. degree in December, 1993.